Debugging a Program With dbx

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-3557-10
May 2000, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Preface 1

1. Starting `dbx` 11

Starting a Debugging Session 11

Debugging an Existing Core File 12

Using the Process ID 12

The `dbx` Start-up Sequence 13

Setting Startup Properties 13

Mapping With the `pathmap` Command 14

Setting Environment Variables With the `dbxenv` Command 14

Creating Your Own dbx Commands Using the `alias` Command 15

Compiling a Program for Debugging 15

Debugging Optimized Code 16

Code Compiled Without the `-g` Option 16

Shared Libraries Require the `-g` Option for Full `dbx` Support 17

Completely Stripped Programs 17

Quitting Debugging 17

Stopping a Process Execution 17

Detaching a Process From `dbx` 18

Killing a Program Without Terminating the Session 18

Saving and Restoring a Debugging Run 18

Using the `save` Command 19

Saving a Series of Debugging Runs as Checkpoints 20

Restoring a Saved Run 20

Saving and Restoring Using `replay` 21

2. Customizing `dbx` 23

Using the `.dbxrc` File 23

Creating a `.dbxrc` File 24

Initialization File Sample 24

The `dbx` Environment Variables and the Korn Shell 25

Customizing `dbx` in Sun WorkShop 25

Setting Debugging Options 25

Maintaining a Unified Set of Options 25

Maintaining Two Sets of Options 26

Storing Custom Buttons 26

Setting `dbx` Environment Variables With the `dbxenv` Command 27

3. Viewing and Visiting Code 33

Mapping to the Location of the Code 33

Visiting Code 34

Visiting a File 34

Visiting Functions 35

Printing a Source Listing 36

Walking the Call Stack to Visit Code 36

Qualifying Symbols With Scope Resolution Operators 36

Backquote Operator 37

C++ Double Colon Scope Resolution Operator 37

Block Local Operator 38

Linker Names 38

Scope Resolution Search Path 38

Locating Symbols 39

Printing a List of Occurrences of a Symbol 39

Determining Which Symbol `dbx` Uses 40

Viewing Variables, Members, Types, and Classes 41

Looking Up Definitions of Variables, Members, and Functions 41

Looking Up Definitions of Types and Classes 42

Using the Auto-Read Facility 45

Debugging Without the Presence of `.o` Files 46

Listing Debugging Information for Modules 47

Listing Modules 47

4. Controlling Program Execution 49

Running a Program 49

Attaching `dbx` to a Running Process 50

Detaching `dbx` From a Process 51

Stepping Through a Program 51

Single Stepping 52

Continuing Execution of a Program 52

Calling a Function 54

Using Ctrl+C to Stop a Process 55

5. Setting Breakpoints and Traces 57

Setting Breakpoints 57

Setting a `stop` Breakpoint at a Line of Source Code 58

Setting a `stop` Breakpoint in a Function 58

Setting a `when` Breakpoint at a Line 59

Setting a Breakpoint in a Dynamically Linked Library 60

Setting Multiple Breaks in C++ Programs 60

Tracing Code 62

Setting a Trace 62

Controlling the Speed of a Trace 63

Listing and Clearing Event Handlers 63

Listing Breakpoints and Traces 63

Deleting Specific Breakpoints Using Handler ID Numbers 63

Watchpoints 64

Faster `modify` Event 66

Setting Breakpoint Filters 66

Efficiency Considerations 67

6. Event Management 69

Event Handlers 69

Creating Event Handlers 70

Manipulating Event Handlers 71

Using Event Counters 71

Setting Event Specifications 71

Breakpoint Event Specifications 72

Watchpoint Event Specifications 73

System Event Specifications 74

Execution Progress Event Specifications 77

Other Event Specifications 78

Event Specification Modifiers 81

Parsing and Ambiguity 83

Using Predefined Variables 83

Variables Valid for `when` Command 84

Variables Valid for Specific Events 85
Setting Event Handler Examples 86
Setting a Watchpoint for Store to an Array Member 86
Implementing a Simple Trace 87
Enabling a Handler While Within a Function (`in` *func*) 87
Determining the Number of Lines Executed 87
Determining the Number of Instructions Executed by a Source Line 88
Enabling a Breakpoint After An Event Occurs 88
Resetting Application Files for `replay` 89
Checking Program Status 89
Catch Floating Point Exceptions 89

7. Using the Call Stack 91
Finding Your Place on the Stack 91
Walking the Stack and Returning Home 92
Moving Up and Down the Stack 92
Moving Up the Stack 92
Moving Down the Stack 93
Moving to a Specific Frame 93
Popping the Call Stack 94
Hiding Stack Frames 94

8. Evaluating and Displaying Data 97
Evaluating Variables and Expressions 97
Verifying Which Variable `dbx` Uses 97
Variables Outside the Scope of the Current Function 98
Printing the Value of a Variable or an Expression 98
Printing C++ 98
Dereferencing Pointers 100

Monitoring Expressions 100
Turning Off Display (Undisplay) 101
Assigning a Value to a Variable 101
Evaluating Arrays 102
Array Slicing 102
Slices 106
Strides 106

9. Using Runtime Checking 109
Capabilities of Runtime Checking 109
When to Use Runtime Checking 110
Runtime Checking Requirements 110
Limitations 111
Using Runtime Checking 111
Turning On Memory Use and Memory Leak Checking 111
Turning On Memory Access Checking 111
Turning On All Runtime Checking 112
Turning Off Runtime Checking 112
Running Your Program 112
Using Access Checking (SPARC only) 116
Understanding the Memory Access Error Report 117
Memory Access Errors 117
Using Memory Leak Checking 118
Detecting Memory Leak Errors 119
Possible Leaks 119
Checking for Leaks 120
Understanding the Memory Leak Report 121
Fixing Memory Leaks 124

Using Memory Use Checking 124
Suppressing Errors 126
Types of Suppression 126
Suppressing Error Examples 127
Default Suppressions 128
Using Suppression to Manage Errors 128
Using Runtime Checking on a Child Process 129
Using Runtime Checking on an Attached Process 133
Using Fix and Continue With Runtime Checking 134
Runtime Checking Application Programming Interface 136
Using Runtime Checking in Batch Mode 136
`bcheck` Syntax 136
`bcheck` Examples 137
Enabling Batch Mode Directly From `dbx` 138
Troubleshooting Tips 138
Runtime Checking's 8 Megabyte Limit 138
Runtime Checking Errors 140
Access Errors 140
Memory Leak Errors 144

10. Data Visualization 147
Specifying Proper Array Expressions 147
Graphing an Array 149
Getting Ready 149
Multiple Ways to Graph an Array 149
Automatic Updating of Array Displays 150
Changing Your Display 151
Analyzing Visualized Data 155

Scenario 1: Comparing Different Views of the Same Data 155

Scenario 2: Updating Graphs of Data Automatically 156

Scenario 3: Comparing Data Graphs at Different Points in a Program 157

Scenario 4: Comparing Data Graphs from Different Runs of the Same Program 157

Fortran Program Example 159

C Program Example 160

11. Fixing and Continuing 161

Using Fix and Continue 161

How fix and continue Operates 162

Modifying Source Using Fix and Continue 162

Fixing Your Program 163

Continuing After Fixing 163

Changing Variables After Fixing 165

Modifying a Header File 166

Fixing C++ Template Definitions 167

12. Debugging Multithreaded Applications 169

Understanding Multithreaded Debugging 169

Thread Information 170

Viewing the Context of Another Thread 171

Viewing the Threads List 172

Resuming Execution 172

Understanding LWP Information 173

13. Debugging Child Processes 175

Attaching to Child Processes 175

Following the `exec` Function 176

Following the `fork` Function 176

Interacting with Events 176

14. Working With Signals 177

Understanding Signal Events 177

Catching Signals 179

Changing the Default Signal Lists 179

Trapping the FPE Signal 179

Sending a Signal in a Program 181

Automatically Handling Signals 181

15. Debugging C++ 183

Using `dbx` with C++ 183

Exception Handling in `dbx` 184

Commands for Handling Exceptions 184

Examples of Exception Handling 186

Debugging With C++ Templates 188

Template Example 189

Commands for C++ Templates 190

16. Debugging Fortran Using `dbx` 197

Debugging Fortran 197

Current Procedure and File 197

Uppercase Letters 198

Optimized Programs 198

Sample `dbx` Session 199

Debugging Segmentation Faults 202

Using `dbx` to Locate Problems 203

Locating Exceptions 204

Tracing Calls 204

Working With Arrays 205

Fortran 95 Allocatable Arrays 207

Showing Intrinsic Functions 210

Showing Complex Expressions 211

Showing Logical Operators 212

Viewing Fortran 95 Derived Types 213

Pointer to Fortran 95 Derived Type 215

17. Debugging at the Machine-Instruction Level 217

Examining the Contents of Memory 217

Using the `examine` or `x` Command 218

Using the `dis` Command 221

Using the `listi` Command 221

Stepping and Tracing at Machine-Instruction Level 222

Single Stepping at the Machine-Instruction Level 222

Tracing at the Machine-Instruction Level 223

Setting Breakpoints at the Machine-Instruction Level 224

Setting a Breakpoint at an Address 225

Using the `adb` Command 225

Using the `regs` Command 225

Platform-Specific Registers 226

Intel Register Information 227

18. Using dbx With the Korn Shell 231

ksh-88 Features Not Implemented 231

Extensions to ksh-88 232

Renamed Commands 232

19. Debugging Shared Libraries 233

Dynamic Linker 233

Link Map 233

Startup Sequence and `.init` Sections 234

Procedure Linkage Tables 234

Debugging Support for Preloaded Shared Objects 234

Fix and Continue 235

Setting a Breakpoint in a Dynamically Linked Library 235

A. Modifying a Program State 237

Impacts of Running a Program Under `dbx` 237

Commands That Alter the State of the Program 238

`assign` Command 238

`pop` Command 239

`call` Command 239

`print` Command 239

`when` Command 240

`fix` Command 240

`cont at` Command 240

Index 241

Preface

`dbx` is an interactive, source-level, command-line debugging tool. *Debugging a Program With dbx* is intended for programmers with a working knowledge of Fortran, C, or C++, and some understanding of the Solaris™ operating environment and UNIX® commands, who want to debug an application using `dbx` commands. It includes references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.

Multiplatform Release

This Sun WorkShop release supports versions 2.6, 7, and 8 of the Solaris™ *SPARC™ Platform Edition* and Solaris *Intel Platform Edition* Operating Environments.

Note – The term "x86" refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture, whereas "*Intel Platform Edition*" appears in the product name.

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your `PATH` environment variable:

1. **Display the current value of the `PATH` variable by typing:**

```
% echo $PATH
```

2. **Review the output for a string of paths containing `/opt/SUNWspro/bin/`.**

   If you find the paths, your `PATH` variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your `PATH` environment variable by following the instructions in this section.

To determine if you need to set your `MANPATH` environment variable:

1. **Request the `workshop` man page by typing:**

```
% man workshop
```

2. **Review the output, if any.**

   If the `workshop`(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in this section for setting your `MANPATH` environment variable.

Note – The information in this section assumes that your Sun WorkShop 6 products were installed in the `/opt` directory. Contact your system administrator if your Sun WorkShop software is not installed in `/opt`.

The `PATH` and `MANPATH` variables should be set in your home `.cshrc` file if you are using the C shell or in your home `.profile` file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your `PATH` variable:

  `/opt/SUNWspro/bin`

- To access Sun WorkShop man pages with the `man` command, add the following to your `MANPATH` variable:

  ```
  /opt/SUNWspro/man
  ```

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` and `MANPATH` variables to access this release, see the *Sun WorkShop 6 Installation Guide* or your system administrator.

How This Book Is Organized

Debugging a Program With dbx contains the following chapters and appendixes:

Chapter 1 "Starting `dbx`" describes how to start a debugging session, discusses compilation options, and tells you how to save all or part of session and replay it later.

Chapter 2 "Customizing `dbx`" describes how to set `dbx` environment variables to customize your debugging environment and how to use the initialization file, `.dbxrc`, to preserve changes and adjustments from session to session.

Chapter 3 "Viewing and Visiting Code" tells you about visiting source files and functions; locating symbols; and looking up variables, members, types, and classes.

Chapter 4 "Controlling Program Execution" describes how to run, attach to, detach from,. continue execution of, stop, and rerun a program under `dbx`. It also tells you how to single-step through program code.

Chapter 5 "Setting Breakpoints and Traces" describes how to set, clear, and list breakpoints and traces, and how to use watchpoints.

Chapter 6 "Event Management" tells you how to manage events, and describes how `dbx` can perform specific actions when specific events occur in the program you are debugging.

Chapter 7 "Finding Your Place on the Stack" tells you how to examine the call stack and how to debug a core file.

Chapter 8 "Evaluating and Displaying Data" shows you how to evaluate data; display the values of expressions, variables, and other data structures; and assign values to variables.

Chapter 9 "Using Runtime Checking" describes how to use runtime checking to detect memory leak and memory access errors in your program automatically.

Chapter 10 “Data Visualization” tells you how to display your data graphically as you debug your program.

Chapter 11 “Fixing and Continuing” describes the fix and continue feature of dbx that allows you to modify and recompile a source file and continue executing without rebuilding your entire program.

Chapter 12 “Debugging Multithreaded Applications” tells you how to find information about threads.

Chapter 13 “Debugging Child Processes” describes several `dbx` facilities that help you debug child processes.

Chapter 14 “Working With Signals” tells you how to use `dbx` to work with signals.

Chapter 15 “Debugging C++” describes dbx support of C++ templates, and the commands available for handling C++ exceptions and how `dbx` handles these exceptions.

Chapter 16 “Debugging Fortran Using `dbx`” introduces some of the `dbx` facilities you can use to debug a Fortran program.

Chapter 17 “Debugging at the Machine-Instruction Level” tells you how to use event management and execution control command at the machine-instruction level, how to display the contents of memory at specific addresses, and how to display source code lines along with their corresponding machine instructions.

Chapter 18 “Using dbx With the Korn Shell” explains the differences between ksh-88 and `dbx` commands.

Chapter 19 “Debugging Shared Libraries” describes `dbx` support for program that use dynamically linked, shared libraries.

Appendix A “Modifying a Program State” focuses on `dbx` commands that change your program or its behavior when you run it under `dbx`.

Typographic Conventions

TABLE P-1 shows the typographic conventions that are used in Sun WorkShop documentation.

TABLE P-1 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Related Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the `docs.sun.com`[sm] Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at the following Web site:

  `http://docs.sun.com`

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:

  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.
  - In your Netscape™ Communicator 4.0 or compatible version browser, open the following file:

    `/opt/SUNWspro/docs/index.html`

  (Contact your system administrator if your Sun WorkShop software is not installed in the `/opt` directory.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open a document in the index, click the document's title.

TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection

Document Collection	Document Title	Description
Forte™ Developer 6 / Sun WorkShop 6 Release Documents	*About Sun WorkShop 6 Documentation*	Describes the documentation available with this Sun WorkShop release and how to access it.
	What's New in Sun WorkShop 6	Provides information about the new features in the current and previous release of Sun WorkShop.
	Sun WorkShop 6 Release Notes	Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.
Forte Developer 6 / Sun WorkShop 6	*Analyzing Program Performance With Sun WorkShop 6*	Explains how to use the new Sampling Collector and Sampling Analyzer (with examples and a discussion of advanced profiling topics) and includes information about the command-line analysis tool `er_print`, the LoopTool and LoopReport utilities, and UNIX profiling tools `prof`, `gprof`, and `tcov`.
	Introduction to Sun WorkShop	Acquaints you with the basic program development features of the Sun WorkShop integrated programming environment.
Forte™ C 6 / Sun WorkShop 6 Compilers C	*C User's Guide*	Describes the C compiler options, Sun-specific capabilities such as pragmas, the `lint` tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
Forte™ C++ 6 / Sun WorkShop 6 Compilers C++	*C++ Library Reference*	Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.
	C++ Migration Guide	Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.
	C++ Programming Guide	Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.
	C++ User's Guide	Provides information on command-line options and how to use the compiler.
	Sun WorkShop Memory Monitor User's Manual	Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see `/opt/SUNWspro/docs/index.html`) and not at the `docs.sun.com` Web site.
Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	*Fortran Library Reference*	Provides details about the library routines supplied with the Fortran compiler.
	Fortran Programming Guide	Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
	Fortran User's Guide	Provides information on command-line options and how to use the compilers.
	FORTRAN 77 Language Reference	Provides a complete language reference.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
	Interval Arithmetic Programming Reference	Describes the intrinsic `INTERVAL` data type supported by the Fortran 95 compiler.
Forte™ TeamWare 6 / Sun WorkShop TeamWare 6	*Sun WorkShop TeamWare 6 User's Guide*	Describes how to use the Sun WorkShop TeamWare code management tools.
Forte Developer 6/ Sun WorkShop Visual 6	*Sun WorkShop Visual User's Guide*	Describes how to use Visual to create C++ and Java™ graphical user interfaces.
Forte™ / Sun Performance Library 6	*Sun Performance Library Reference*	Discusses the optimized library of subroutines and functions used to perform computational linear algebra and fast Fourier transforms.
	Sun Performance Library User's Guide	Describes how to use the Sun-specific features of the Sun Performance Library, which is a collection of subroutines and functions used to solve linear algebra problems.
Numerical Computation Guide	*Numerical Computation Guide*	Describes issues regarding the numerical accuracy of floating-point computations.
Standard Library 2	*Standard C++ Class Library Reference*	Provides details on the Standard C++ Library.
	Standard C++ Library User's Guide	Describes how to use the Standard C++ Library.
Tools.h++ 7	*Tools.h++ Class Library Reference*	Provides details on the `Tools.h++` class library.
	Tools.h++ User's Guide	Discusses use of the C++ classes for enhancing the efficiency of your programs.

TABLE P-4 describes related Solaris documentation available through the `docs.sun.com` Web site.

TABLE P-4 Related Solaris Documentation

Document Collection	Document Title	Description
Solaris Software Developer	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.
	Programming Utilities Guide	Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.

CHAPTER 1

Starting dbx

dbx is an interactive, source-level, command-line debugging tool. You can use it to run a program in a controlled manner and to inspect the state of a stopped program. dbx gives you complete control of the dynamic execution of a program, including the collection of performance data.

This chapter contains the following sections:

- Starting a Debugging Session
- Setting Startup Properties
- Debugging Optimized Code
- Quitting Debugging
- Saving and Restoring a Debugging Run

Starting a Debugging Session

How you start dbx depends on what you are debugging, where you are, what you need dbx to do, how familiar you are with dbx, and whether or not you have set up any dbx environment variables.

The simplest way to start a dbx session is to type the dbx command at a shell prompt.

```
$ dbx
```

To start dbx from a shell and load a program to be debugged, type:

```
$ dbx program_name
```

For more information on the dbx command and start-up options, see "dbx Command" and "Invoking dbx" in the Using dbx Commands section of the Sun WorkShop™ online help, and the dbx(1) man page.

`dbx` is started automatically when you start debugging a program using the Sun WorkShop Debugging window (see "Debugging the Current Program" and "Debugging a Program New to Sun WorkShop" in the Using the Debugging Window section of the Sun WorkShop online help).

Debugging an Existing Core File

If the program that dumped core was dynamically linked with any shared libraries, it is important to debug the core file in the same operating environment in which it was created.

To debug a core file, type:

```
$ dbx program_name core
```

Use the `where` command to determine where the program was executing when it dumped core.

When you debug a core file, you can also evaluate variables and expressions to see the values they had at the time the program crashed, but you cannot evaluate expressions that make function calls.

For more information, see "Core File Debugging with dbx" in the Using dbx Commands section of the Sun WorkShop online help.

Using the Process ID

You can attach a running process to `dbx` using the *process_ID* (*pid*) as an argument to the `dbx` command.

```
$ dbx your_program_name process_ID
```

You can also attach to a process using its process ID number without knowing the name of the program.

```
$ dbx - process_ID
```

Because the program name remains unknown to `dbx`, you cannot pass arguments to the process in a `run` type command.

To attach dbx to a running process in the Sun WorkShop Debugging window, choose Debug ➤ Attach Process. For more information, see "Attaching to a Running Process" in the Using the Debugging Window section of the Sun WorkShop online help.

The dbx Startup Sequence

Upon invocation, dbx looks for and reads the installation startup file, .dbxrc in the directory *install-directory*/SUNWspro/lib, where the default *install-directory* is /opt.

Next, dbx searches for the startup file .dbxrc in the current directory, then in $HOME. If the file is not found, it searches for the startup file .dbxinit in the current directory, then in $HOME.

Generally, the contents of .dbxrc and .dbxinit files are the same with one major exception. In the .dbxinit file, the alias command is defined as dalias and not the normal default, which is kalias, the alias command for the Korn shell. A different startup file may be specified explicitly using the -s command-line option. For more information, see "Using the .dbxrc File" on page 23 and "Creating a .dbxrc File" in the Using dbx Commands section of the Sun WorkShop online help.

A startup file may contain any dbx command, and commonly contains alias, dbxenv, pathmap, and Korn shell function definitions. However, certain commands require that a program has been loaded or a process has been attached to. All startup files are loaded before the program or process is loaded. The startup file may also source other files using the source or .(period) command. You can also use the startup file to set other dbx options.

As dbx loads program information, it prints a series of messages, such as Reading *filename*.

Once the program is finished loading, dbx is in a ready state, visiting the "main" block of the program (for C, C++, or Fortran 95: main(); for FORTRAN 77: MAIN()). Typically, you set a breakpoint and then issue a run command, such as stop in main and run for a C program.

Setting Startup Properties

You can use the pathmap, dbxenv, and alias commands to set startup properties for your dbx sessions.

Mapping With the `pathmap` Command

By default, `dbx` looks in the directory in which the program was compiled for the source files associated with the program being debugged. If the source or object files are not there or the machine you are using does not use the same path name, you must inform `dbx` of their location.

If you move the source or object files, you can add their new location to the search path. The `pathmap` command creates a mapping from your current view of the file system to the name in the executable image. The mapping is applied to source paths and object file paths.

Add common pathmaps to your `.dbxrc` file.

To establish a new mapping from the directory *from* to the directory *to*, type:

```
(dbx) pathmap [ -c ] from to
```

If `-c` is used, the mapping is applied to the current working directory as well.

The `pathmap` command is useful for dealing with automounted and explicit NFS-mounted file systems with different base paths on differing hosts. Use `-c` when you try to correct problems due to the automounter because current working directories are inaccurate on automounted file systems.

The mapping of `/tmp_mnt` to `/` exists by default.

For more information, see "pathmap Command" in the Using dbx Commands section of the Sun WorkShop online help.

Setting Environment Variables With the `dbxenv` Command

You can use the `dbxenv` command to either list or set `dbx` customization variables. You can place `dbxenv` commands in your `.dbxrc` file. To list variables, type:

```
$ dbxenv
```

You can also set `dbx` environment variables. See Chapter 2 for more information about the `.dbxrc` file and about setting these variables.

For more information, see “Setting dbx Environment Variables With the dbxenv Command” on page 27 and “dbxenv Command” in the Using dbx Commands section of the Sun WorkShop online help.

Creating Your Own dbx Commands Using the alias Command

You can create your own dbx commands using the kalias or dalias commands. For more information, see “kalias Command” and “dalias Command” in the Using dbx Commands section of the Sun WorkShop online help.

Global variables can be printed and assigned to as normal, although they might have inaccurate values if the final register-to-memory store has not yet occurred.

Parameters can only be printed if you are stopped at the very beginning of a function, since the parameter registers can be re-used by the compiler. The values of stack variables cannot be printed or manipulated in optimized programs because the stack and register locations of these variables are not recorded.

The *Analyzing a Program With Sun WorkShop* manual contains more information on compiler optimizations that might be helpful when using the Sun Workshop tools on an optimized program.

Compiling a Program for Debugging

You must prepare your program for debugging with dbx by compiling it with the -g or -g0 option.

The -g option instructs the compiler to generate debugging information during compilation.

For example, to compile using C++, type:

```
% CC -g example_source.cc
```

In C++, the -g option turns on debugging and turns off inlining of functions. The -g0 (zero) option turns on debugging and does not affect inlining of functions. You cannot debug inline functions with the -g0 option. The -g0 option can significantly decrease link time and dbx start-up time (depending on the use of inlined functions by the program).

To compile optimized code for use with `dbx`, compile the source code with both the `-O` (uppercase letter O) and the `-g` options.

Debugging Optimized Code

The `dbx` tool provides partial debugging support for optimized code. The extent of the support depends largely upon how you compiled the program.

When analyzing optimized code, you can:

- Stop execution at the start of any function (`stop in` *function* command)
- Evaluate, display, or modify arguments
- Evaluate, display, or modify global or static variables

However, with optimized code, `dbx` cannot:

- Single-step from one line to another (`next` or `step` command)
- Evaluate, display, or modify local variables

When programs are compiled with optimization and debugging enabled at the same time (using the `-O -g` options), `dbx` operates in a restricted mode.

Source line information is available, but the code for one source line might appear in several different places for an optimized program, so stepping through a program by source line results in the "current line" jumping around in the source file, depending on how the code was scheduled by the optimizer.

Tail call optimization can result in missing stack frames when the last effective operation in a function is a call to another function.

Generally, symbolic information for parameters, locals, and globals is available for optimized programs. Type information about structs, unions, C++ classes, and the types and names of locals, globals, and parameters should be available. Complete information about the location of these items in the program is not available for optimized programs.

Code Compiled Without the `-g` Option

While most debugging support requires that a program be compiled with `-g`, `dbx` still provides the following level of support for code compiled without `-g`:

- Backtrace (`dbx where` command)
- Calling a function (but without parameter checking)
- Checking global variables

Note, however, that `dbx` cannot display source code unless the code was compiled with the `-g` option. This restriction also applies to code that has had `strip -x` applied to it.

Shared Libraries Require the `-g` Option for Full `dbx` Support

For full support, a shared library must also be compiled with the `-g` option. If you build a program with shared library modules that were not compiled with the `-g` option, you can still debug the program. However, full `dbx` support is not possible because the information was not generated for those library modules.

Completely Stripped Programs

The `dbx` tool can debug programs that have been completely stripped. These programs contain some information that can be used to debug your program, but only externally visible functions are available. Runtime checking cannot work on stripped programs or load objects.

Quitting Debugging

A `dbx` session runs from the time you start `dbx` until you quit `dbx`; you can debug any number of programs in succession during a `dbx` session.

To quit a `dbx` session, type `quit` at the `dbx` prompt.

```
(dbx) quit
```

When you start `dbx` and attach it to a running process using the *process_id* option, the process survives and continues when you quit the debugging session. `dbx` performs an implicit `detach` before quitting the session.

Stopping a Process Execution

You can stop execution of a process at any time by pressing Ctrl+C without leaving `dbx`.

Detaching a Process From dbx

If you have attached dbx to a process, you can detach the process from dbx without killing it or the dbx session by using the detach command.

To detach a process from dbx without killing the process, type:

```
(dbx) detach
```

For more information, see "detach Command" in the Using dbx Commands section of the Sun WorkShop online help.

Killing a Program Without Terminating the Session

The dbx kill command terminates debugging of the current process as well as killing the process. However, kill preserves the dbx session itself leaving dbx ready to debug another program.

Killing a program is a good way of eliminating the remains of a program you were debugging without exiting dbx.

To kill a program executing in dbx, type:

```
(dbx) kill
```

For more information, see "kill Command" in the Using dbx Commands section of the Sun WorkShop online help.

Saving and Restoring a Debugging Run

The dbx tool provides three commands for saving all or part of a debugging run and replaying it later:

- save [-*number*] [*filename*]
- restore [*filename*]
- replay [-*number*]

Using the `save` Command

The `save` command saves to a file all debugging commands issued from the last `run`, `rerun`, or `debug` command up to the `save` command. This segment of a debugging session is called a *debugging run*.

The `save` command saves more than the list of debugging commands issued. It saves debugging information associated with the state of the program at the start of the run—breakpoints, display lists, and the like. When you restore a saved run, `dbx` uses the information in the save-file.

You can save part of a debugging run; that is, the whole run minus a specified number of commands from the last one entered. Example A shows a complete saved run. Example B shows the same run saved, minus the last two steps.

Example A: Saving a complete run

```
debug
stop at line
run
next
next
stop at line
continue
next
next
step
next
save
```

Example B: Saving a run minus the last two steps

```
debug
stop at line
run
next
next
stop at line
continue
next
next
step
next
save-2
```

If you are not sure where you want to end the run you are saving, use the `history` command to see a list of the debugging commands issued since the beginning of the session.

Note – By default, the `save` command writes information to a special save-file. If you want to save a debugging run to a file you can restore later, you can specify a file name with the `save` command. See "Saving a Series of Debugging Runs as Checkpoints" on page 20.

To save an entire debugging run up to the `save` command, type:

```
(dbx) save
```

To save part of a debugging run, use the `save` *number* command, where *number* is the number of commands back from the `save` command that you do not want saved.

```
(dbx) save number
```

Saving a Series of Debugging Runs as Checkpoints

If you save a debugging run without specifying a file name, `dbx` writes the information to a special save-file. Each time you save, `dbx` overwrites this save-file. However, by giving the `save` command a *filename* argument, you can save a debugging run to a file that you can restore later, even if you have saved other debugging runs since the one saved to *filename*.

Saving a series of runs gives you a set of *checkpoints*, each one starting farther back in the session. You can restore any one of these saved runs, continue, then reset `dbx` to the program location and state saved in an earlier run.

To save a debugging run to a file other than the default save-file:

```
(dbx) save filename
```

Restoring a Saved Run

After saving a run, you can restore the run using the `restore` command. `dbx` uses the information in the save-file. When you restore a run, `dbx` first resets the internal state to what it was at the start of the run, then reissues each of the debugging commands in the saved run.

Note – The `source` command also reissues a set of commands stored in a file, but it does not reset the state of `dbx`; it only reissues the list of commands from the current program location.

Prerequisites for an Exact Restoration of a Saved Run

For exact restoration of a saved debugging run, all the inputs to the run must be exactly the same: arguments to a `run`-type command, manual inputs, and file inputs.

Note – If you save a segment and then issue a `run`, `rerun`, or `debug` command before you do a `restore`, `restore` uses the arguments to the second, post-save `run`, `rerun`, or `debug` command. If those arguments are different, you might not get an exact restoration.

To restore a saved debugging run:, type:

```
(dbx) restore
```

To restore a debugging run saved to a file other than the default save-file:, type:

```
(dbx) restore filename
```

Saving and Restoring Using `replay`

The `replay` command is a combination command, equivalent to issuing a `save -1` followed immediately by a `restore`. The `replay` command takes a negative *number_of_ commands* argument, which it passes to the `save` portion of the command. By default, the value of *-number* is `-1`, so `replay` works as an undo command, restoring the last run up until, but not including, the last command issued.

To replay the current debugging run, minus the last debugging command issued, type:

```
(dbx) replay
```

To replay the current debugging run and stop the run before a specific command, use the `dbx replay` command, where *number* is the number of commands back from the last debugging command.

```
(dbx) replay -number
```

CHAPTER 2

Customizing dbx

This chapter describes the dbx environment variables you can use to customize certain attributes of your debugging environment, and how to use the initialization file, .dbxrc, to preserve your changes and adjustments from session to session.

This chapter is organized into the following sections:

- Using the .dbxrc File
- The dbx Environment Variables and the Korn Shell
- Customizing dbx in Sun WorkShop
- Setting dbx Environment Variables With the dbxenv Command

Using the .dbxrc File

The dbx initialization file, .dbxrc, stores dbx commands that are executed each time you start dbx. Typically, the file contains commands that customize your debugging environment, but you can place any dbx commands in the file. If you customize dbx from the command line while you are debugging, those settings apply only to the current debugging session.

Note – A .dbxrc file should not contain commands that execute your code. However, you can put such commands in a file, and then use the dbx source command to execute the commands in that file.

During startup, dbx searches for .dbxrc first. The search order is:

1. Current directory ./.dbxrc
2. Home directory $HOME/.dbxrc

If .dbxrc is not found, dbx prints a warning message and searches for .dbxinit (dbx mode). The search order is:

1. Current directory ./.dbxinit

2. Home directory `$HOME/.dbxinit`

Creating a `.dbxrc` File

To suppress the warning message and create a `.dbxrc` file that contains common customizations and aliases, type in the command pane:

```
help .dbxrc>$HOME/.dbxrc
```

You can then customize the resulting file by using your text editor to uncomment the entries you want to have executed.

Initialization File Sample

Here is a sample `.dbxrc` file:

```
dbxenv input_case_sensitive false
catch FPE
```

The first line changes the default setting for the case sensitivity control:

- `dbxenv` refers to the set of debugging environment attributes.
- `input_case_sensitive` refers to the matching control.
- `false` is the control setting.

The next line is a debugging command, `catch`, which adds a system signal, `FPE` to the default list of signals to which `dbx` responds, stopping the program.

For more information, see “Typical Entries in a .dbxrc File,” “Useful Aliases in a .dbxrc File,”, and “Useful Functions in a .dbxrc File” in the Using dbx Commands section of the Sun WorkShop online help.

The dbx Environment Variables and the Korn Shell

Each dbx environment variable is also accessible as a ksh variable. The name of the ksh variable is derived from the dbx environment varaible by prefixing it with DBX_. For example dbxenv stack_verbose and echo $DBX_stack_verbose yield the same output.

Customizing dbx in Sun WorkShop

If you use dbx through the Dbx Commands window in Sun WorkShop Debugging as well as from the command line in a shell, you can customize dbx most effectively by taking advantage of the customization features in Sun WorkShop Debugging.

Setting Debugging Options

If you use dbx primarily in the Dbx Commands window in Sun WorkShop Debugging, you should set most dbx environment variables using the Debugging Options dialog box. See the "Debugging Options Dialog Box" in the Using the Debugging Window section in the Sun WorkShop online help.

When you set debugging options using the Debugging Options dialog box, the dbxenv commands for the corresponding environment variables are stored in the Sun WorkShop configuration file .workshoprc. When you start to debug a program in the Sun WorkShop Debugging window, any settings in your .dbxrc file that conflict with those in your .workshoprc take precedence.

Maintaining a Unified Set of Options

If you use dbx both from the command line in a shell and from within Sun WorkShop, you can create a unified set of options that customizes dbx for both modes.

To create a unified set of options:

1. **In the Sun WorkShop Debugging window, choose Debug ➤ Debugging Options.**

2. **Click Save as Defaults to make your current option settings your defaults.**
3. **Open your `.dbxrc` file in an editor window.**
4. **Near the top, type the line `source $HOME/.workshoprc`.**
5. **Move the relevant `dbxenv` commands to a location after the source line, or delete them, or comment them out.**
6. **Save the file.**

   Any changes made through the Debugging Options dialog box are now available to `dbx` both when using Sun WorkShop and when running in a shell.

Maintaining Two Sets of Options

To maintain different options settings for running `dbx` within Sun WorkShop and from the command line in a shell, you can use the `havegui` variable in your `.dbxrc` file to conditionalize your `dbxenv` commands. For example:

```
if $havegui
  then
     dbxenv follow_fork_mode ask
     dbxenv stack_verbose on
  else
      dbxenv follow_fork_mode parent
      dbxenv stack_verbose off
```

Storing Custom Buttons

Sun WorkShop Debugging includes the Button Editor for adding, removing, and editing buttons in the Custom Buttons window (and also the Debugging window and editor window tool bars). You no longer need to use the `button` command to store buttons in your `.dbxrc` file. You cannot add buttons to, or remove buttons from, your `.dbxrc` file with the Button Editor.

You cannot permanently delete a button stored in your `.dbxrc` file with the Button Editor. The button will reappear in your next debugging session. You can remove such a button by editing your `.dbxrc` file.

Setting dbx Environment Variables With the dbxenv Command

You can use the `dbxenv` command to set the dbx environment variables that customize your dbx sessions.

To display the value of a specific variable, type:

```
(dbx) dbxenv variable
```

To show all variables and their values, type:

```
(dbx) dbxenv
```

To set the value of a variable, type:

```
(dbx) dbxenv variable value
```

Note – Each variable has a corresponding ksh environment variable such as `DBX_trace_speed`. You can assign the variable directly or with the `dbxenv` command; they are equivalent.

TABLE 2-1 shows all of the dbx environment variables that you can set:

TABLE 2-1 dbx Environment Variables

dbx **Environment Variable**	**What the Variable Does**
`allow_critical_exclusion on|off`	Normally `loadobject -x` disallows the exclusion of certain shared libraries critical to `dbx` functionality. By setting this variable to `on` that restriction is defeated. While this variable is `on` only core files can be debugged. Default: `off`.
`aout_cache_size` *number*	Size of `a.out` loadobject cache; set this to *number* when debugging *number* programs serially from a single `dbx`. A *number* of zero still allows caching of shared objects. See `locache_enable`. Default: 1.
`array_bounds_check on|off`	If set to `on`, `dbx` checks the array bounds. Default: `on`.
`cfront_demangling on|off`	Governs demangling of Cfront (SC 2.0 and SC 2.0.1) names while loading a program. It is necessary to set this parameter to `on` or start `dbx` with the `-F` option if debugging programs compiled with Cfront or programs linked with Cfront compiled libraries. If set to `off`, `dbx` loads the program approximately 15% faster. Default: `off`.
`delay_xs on|off`	Governs whether debugging information for modules compiled with the `-xs` option is loaded at `dbx` startup or delayed. Default: `on`.
`disassembler_version autodetect|v8|v9|v9vis`	SPARC platform: Sets the version of `dbx`'s built-in disassembler for SPARC V8, V9, or V9 with the Visual Instruction set. Default is `autodetect`, which sets the mode dynamically depending on the type of the machine `a.out` is running on. x86 platforms: The valid choice is `autodetect`.
`fix_verbose on|off`	Governs the printing of compilation line during a `fix`. Default: `off`
`follow_fork_inherit on|off`	When following a child, inherits or does not inherit events. Default: `off`
`follow_fork_mode parent|child|both|ask`	Determines which process is followed after a fork; that is, when the current process executes a `fork`, `vfork`, or `fork1`. If set to `parent`, the process follows the parent. If set to `child`, it follows the child. If set to `both`, it follows the child, but the parent process remains active. If set to `ask`, you are asked which process to follow whenever a fork is detected. Default: `parent`.

TABLE 2-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`follow_fork_mode_inner unset| parent|child|both`	Of relevance after a fork has been detected if `follow_fork_mode` was set to `ask`, and you chose `stop`. By setting this variable, you need not use `cont -follow`.
`input_case_sensitive autodetect| true|false`	If set to `autodetect`, `dbx` automatically selects case sensitivity based on the language of the file: `false` for FORTRAN 77 or Fortran 95 files, otherwise `true`. If `true`, case matters in variable and function names; otherwise, case is not significant. Default: `autodetect.`
`language_mode autodetect|main|c| ansic|c++|objc|fortran| fortran90|native_java`	Governs the language used for parsing and evaluating expressions. `autodetect`: sets to the language of the current file. Useful if debugging programs with mixed languages (default). `main`: sets language of the main routine in the program. Useful if debugging homogeneous programs. `c`, `c++`, `ansic`, `c++`, `objc`, `fortran`, `fortran90`, `native_java`: sets to selected language.
`locache_enable on|off`	Enables or disables loadobject cache entirely. Default: `on`.
`mt_scalable on|off`	When enabled, `dbx` will be more conservative in its resource usage and will be able to debug processes with upwards of 300 LWPs. The down side is significant slowdown. Default: `off`.
`output_auto_flush on|off`	Automatically calls `fflush()` after each `call`. Default: `on`
`output_base 8|10|16|automatic`	Default base for printing integer constants. Default: `automatic` (pointers in hexadecimal characters, all else in decimal).
`output_dynamic_type on|off`	When set to `on`, `-d` is the default for printing, displaying, and inspecting. Default: `off`.
`output_inherited_members on|off`	When set to `on`, -r is the default for printing, displaying, and inspecting. Default: `off`.
`output_list_size` *num*	Governs the default number of lines to print in the `list` command. Default: 10.
`output_log_file_name` *filename*	Name of the command logfile. Default: `/tmp/dbx.log.`*uniqueID*
`output_max_string_length` *number*	Sets *number* of characters printed for `char *s`. Default: 512.
`output_pretty_print on|off`	Sets `-p` as the default for printing, displaying, and inspecting. Default: `off`.

TABLE 2-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`output_short_file_name on|off`	Displays short path names for files. Default: `on`.
`overload_function on|off`	For C++, if set to `on`, does automatic function overload resolution. Default: `on`.
`overload_operator on|off`	For C++, if set to `on`, does automatic operator overload resolution. Default: `on`.
`pop_auto_destruct on|off`	If set to `on`, automatically calls appropriate destructors for locals when `pop`ping a frame. Default: `on`.
`proc_exclusive_attach on|off`	If set to `on`, keeps `dbx` from attaching to a process if another tool is already attached. Warning: be aware that if more than one tool attaches to a process and tries to control it chaos ensues. Default: `on`.
`rtc_auto_continue on|off`	Logs errors to `rtc_error_log_file_name` and continue. Default: `off`.
`rtc_auto_suppress on|off`	If set to `on`, an RTC error at a given location is reported only once. Default: `off`.
`rtc_biu_at_exit on|off|verbose`	Used when memory use checking is on explicitly or via `check -all`. If the value is `on`, a non-verbose memory use (blocks in use) report is produced at program exit. If the value is `verbose`, a verbose memory use report is produced at program exit. The value `off` causes no output. Default: `on`.
`rtc_error_limit` *number*	*Number* of RTC errors to be reported. Default: 1000.
`rtc_error_log_file_name` *filename*	Name of file to which RTC errors are logged if `rtc_auto_continue` is set. Default: `/tmp/dbx.errlog.`*uniqueID*
`rtc_error_stack on|Off`	If set to `on`, stack traces show frames corresponding to RTC internal mechanisms. Default: `off`.
`rtc_inherit on|Off`	If set to `on`, enables runtime checking on child processes that are executed from the debugged program and causes `LD_PRELOAD` to be inherited. Default: `off`.
`rtc_mel_at_exit on|off|verbose`	Used when memory leak checking is `on`. If the value is `on`, a non-verbose memory leak report is produced at program exit. If the value is `verbose`, a verbose memory leak report is produced at program exit. The value `off` causes no output. Default: `on`.

TABLE 2-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`rtc_use_traps on|off`	Used to enable a work around for the eight megabyte code limitation on runtime checking. If set to `on`, it causes `dbx` to use UltraSparc trap instructions when they are available. Default: `off`.
`run_autostart on|off`	If set to `on` with no active program, `step`, `next`, `stepi`, and `nexti` implicitly run the program and stop at the language-dependent `main` routine. If set to `on`, `cont` implies `run` when necessary. Default: `off`.
`run_io stdio|pt`	Governs whether the user program's input/output is redirected to `dbx`'s `stdio` or a specific pty. The pty is provided by `run_pty`. Default: `stdio`.
`run_pty` *ptyname*	Sets the name of the pty to use when `run_io` is set to `pty`. Ptys are used by graphical user interface wrappers.
`run_quick on|off`	If set to `on`, no symbolic information is loaded. The symbolic information can be loaded on demand using `prog -readsysms`. Until then, `dbx` behaves as if the program being debugged is stripped. Default: `off`.
`run_savetty on | off`	Multiplexes tty settings, process group, and keyboard settings (if `-kbd` was used on the command line) between `dbx` and the program being debugged. Useful when debugging editors and shells. Set to `on` if `dbx` gets `SIGTTIN` or `SIGTTOU` and pops back into the shell. Set to `off` to gain a slight speed advantage. The setting is irrelevant if `dbx` is attached to the program being debugged or is running under Sun WorkShop. Default: `on`.
`run_setpgrp on | off`	If set to `on`, when a program is `run` `setpgrp(2)`is called right after the fork. Default: `off`.
`scope_global_enums on |off`	If set to `on`, enumerators are put in global scope and not in file scope. Set before debugging information is processed (`~/.dbxrc`). Default: `off`.
`scope_look_aside on | off`	If set to `on`, finds file static symbols, even when not in scope. Default: on.
`session_log_file_name` *filename*	Name of the file where `dbx` logs all commands and their output. Output is appended to the file. Default: "" (no session logging).

TABLE 2-1 dbx Environment Variables *(Continued)*

dbx **Environment Variable**	**What the Variable Does**
`stack_find_source on|off`	When set to `on`, `dbx` attempts to find and automatically pop up to a stack frame with source when the program being debugged comes to a stop in a function that is not compiled with `-g`. Default: `on`.
`stack_max_size` *number*	Sets the default size for the `where` command. Default: 100.
`stack_verbose on|off`	Governs the printing of arguments and line information in `where`. Default: `on`.
`step_events on|off`	When set to `on`, allows breakpoints while stepping and nexting. Default: `off`.
`step_granularity statement	line`	Controls granularity of source line stepping. When set to `statement` the following code: `a(); b();` takes the two next commands to execute. When set to `line` a single next command executes the code. The granularity of line is particularly useful when dealing with multiline macros. Default: `statement`.
`suppress_startup_message` n*umber*	Sets the release level below which the startup message is not printed. Default: 3.01.
`symbol_info_compression on|off`	When set to `on`, reads debugging information for each `include` file only once. Default: `on`.
`trace_speed` *number*	Sets the speed of tracing execution. Value is the number of seconds to pause between steps. Default: 0.50.

CHAPTER 3

Viewing and Visiting Code

Each time the program you are debugging stops, `dbx` prints the source line associated with the *stop location*. At each program stop, `dbx` resets the value of the *current function* to the function in which the program is stopped. Before the program starts running and when it is stopped, you can move to, or visit, functions and files elsewhere in the program.

This chapter describes how `dbx` navigates through code and locates functions and symbols. It also covers how to use commands to visit code or look up declarations for identifiers, types, and classes.

This chapter is organized into the following sections

- Mapping to the Location of the Code
- Visiting Code
- Qualifying Symbols With Scope Resolution Operators
- Locating Symbols
- Viewing Variables, Members, Types, and Classes
- Using the Auto-Read Facility

Mapping to the Location of the Code

`dbx` must know the location of the source and object code files associated with a program. The default directory for the object files is the directory the files were in when the program was last linked. The default directory for the source files is the one they were in when last compiled. If you move the source or object files, or copy them to a new location, you must either relink the program or change to the new location before debugging.

If you move the source or object files, you can add their new location to the search path. The `pathmap` command creates a mapping from your current view of the file system to the name in the executable image. The mapping is applied to source paths and object file paths.

To establish a new mapping from the directory *from* to the directory *to*:

```
(dbx) pathmap [-c] from to
```

If `-c` is used, the mapping is applied to the current working directory as well.

The `pathmap` command is also useful for dealing with automounted and explicit NFS mounted file systems with different base paths on differing hosts. Use `-c` when you try to correct problems due to the automounter because current working directories are inaccurate on automounted file systems.

The mapping of `/tmp_mnt` to `/` exists by default.

For more information, see "pathmap Command" in the Using dbx Commands section of the Sun WorkShop online help.

Visiting Code

You can visit code elsewhere in the program any time the program is not running. You can visit any function or file that is part of the program.

Visiting a File

You can visit any file `dbx` recognizes as part of the program (even if a module or file was not compiled with the `-g` option). Visiting a file does not change the current function. To visit a file:

```
(dbx) file filename
```

Using the `file` command by itself echoes the file you are currently visiting.

```
(dbx) file
```

`dbx` displays the file from its first line unless you specify a line number.

```
(dbx) file filename ; list line_number
```

Visiting Functions

You can use the `func` command to visit a function. To visit a function, type the command `func` followed by the function name. For example:

```
(dbx) func adjust_speed
```

The `func` command by itself echoes the currently visited function.

Selecting From a List of C++ Ambiguous Function Names

If you try to visit a C++ member function with an ambiguous name or an overloaded function name, a list is displayed, showing all functions with the overloaded name.

If the specified function is ambiguous, type the number of the function you want to visit. If you know which specific class a function belongs to, you can type:

```
(dbx) func block::block
```

Choosing Among Multiple Occurrences

If multiple symbols are accessible from the same scope level, `dbx` prints a message reporting the ambiguity.

```
(dbx) func main
(dbx) which C::foo
More than one identifier 'foo'.
Select one of the following:
 0) Cancel
 1) 'a.out't.cc'C::foo(int)
 2) 'a.out't.cc'C::foo()
>1
'a.out't.cc'C::foo(int)
```

In the context of the `which` command, choosing from the list of occurrences does not affect the state of `dbx` or the program. Whichever occurrence you choose, `dbx` echoes the name.

The `which` command tells you which symbol `dbx` would choose. In the case of ambiguous names, the overload display list indicates that `dbx` has not yet determined which occurrence of two or more names it would use. `dbx` lists the possibilities and waits for you to choose one.

For more information, see "func Command" in the Using dbx commands section of the Sun WorkShop online help.

Printing a Source Listing

Use the `list` command to print the source listing for a file or function. Once you visit a file, `list` prints *number* lines from the top. Once you visit a function, `list` prints its lines.

```
(dbx) list [-i | -instr] [+] [-] number [ function | filename ]
```

For more information on the `list` command, see "list Command" in the Using dbx Commands section of the Sun WorkShop online help.

Walking the Call Stack to Visit Code

Another way to visit code when a live process exists is to "walk the call stack," using the stack commands to view functions currently on the stack.

Walking the stack causes the current function and file to change each time you display a stack function. The stop location is considered to be at the "bottom" of the stack, so to move away from it, use the `up` command, that is, move toward the `main` or `begin` function. Use the `down` command to move toward the current frame.

For more information on walking the call stack, see "Walking the Stack and Returning Home" on page 92.

Qualifying Symbols With Scope Resolution Operators

When using the `func` or `file` command, you might need to use *scope resolution operators* to qualify the names of the functions that you give as targets.

dbx provides three scope resolution operators with which to qualify symbols: the backquote operator (‘), the C++ double colon operator (::), and the block local operator (:*lineno*). You use them separately or, in some cases, together.

In addition to qualifying file and function names when visiting code, symbol name qualifying is also necessary for printing and displaying out-of-scope variables and expressions, and for displaying type and class declarations (whatis command). The symbol qualifying rules are the same in all cases; this section covers the rules for all types of symbol name qualifying.

Backquote Operator

Use the backquote character (‘) to find a variable of global scope:

```
(dbx) print ‘item
```

A program can use the same function name in two different files (or compilation modules). In this case, you must also qualify the function name to dbx so that it registers which function you will visit. To qualify a function name with respect to its file name, use the general purpose backquote (‘) scope resolution operator.

```
(dbx) func ‘file_name‘function_name
```

C++ Double Colon Scope Resolution Operator

Use the double colon operator (::) to qualify a C++ member function or top level function with:

- An overloaded name (same name used with different argument types)
- An ambiguous name (same name used in different classes)

You might want to qualify an overloaded function name. If you do not qualify it, dbx displays an overload list so you can choose which function you will visit. If you know the function class name, you can use it with the double colon scope resolution operator to qualify the name.

```
(dbx) func class::function_name (args)
```

For example, if `hand` is the class name and `draw` is the function name, type:

```
(dbx) func hand::draw
```

Block Local Operator

The block local operator (:*lineno*) is used in conjunction with the backquote operator. It identifies the line number of an expression that references the instance you're interested in.

In the following example, `:230` is the block local operator.

```
(dbx) stop in `animate.o`change_glyph:230`item
```

Linker Names

`dbx` provides a special syntax for looking up symbols by their linker names (mangled names in C++). Prefix the symbol name with a # (pound sign) character (use the ksh escape character \ (backslash) before any $ (dollar sign) characters), as in these examples:

```
(dbx) stop in #.mul
(dbx) whatis #\$FEcopyPc
(dbx) print `foo.c`#staticvar
```

Scope Resolution Search Path

When you issue a debugging command with a *symbol* target name, the search order is as follows:

1. Within the scope of the current function. If the program is stopped in a nested block, `dbx` searches within that block, then in the scope of all enclosing blocks.
2. For C++ only: class members of the current function's class and its base class.
3. For C++ only: the current name space.
4. The immediately enclosing "compilation unit," generally, the file containing the current function.

5. The LoadObject scope.
6. The global scope.
7. If none of the above searches are successful, `dbx` assumes you are referencing a private, or file static, variable or function. `dbx` optionally searches for a file static symbol in every compilation unit depending on the value of the `dbxenv` setting `scope_look_aside`.

`dbx` uses whichever occurrence of the symbol it first finds along this search path. If `dbx` cannot find a variable, it reports an error.

Locating Symbols

In a program, the same name might refer to different types of program entities and occur in many scopes. The `dbx whereis` command lists the fully qualified name—hence, the location—of all symbols of that name. The `dbx which` command tells you which occurrence of a symbol `dbx` uses if you give that name as the target of a debugging command.

Printing a List of Occurrences of a Symbol

To print a list of all the occurrences of a specified symbol, use `whereis` *symbol,* where *symbol* can be any user-defined identifier. For example:

```
(dbx) whereis table
forward: `Blocks`block_draw.cc`table
function: `Blocks`block.cc`table::table(char*, int, int, const
point&)
class: `Blocks`block.cc`table
class: `Blocks`main.cc`table
variable:       `libc.so.1`hsearch.c`table
```

The output includes the name of the loadable object(s) where the program defines *symbol*, as well as the kind of entity each object is: class, function, or variable.

Because information from the `dbx` symbol table is read in as it is needed, the `whereis` command registers only occurrences of a symbol that are already loaded. As a debugging session gets longer, the list of occurrences can grow.

For more information, see “whereis Command” in the Using dbx Commands section of the Sun WorkShop online help.

Determining Which Symbol dbx Uses

The which command tells you which symbol with a given name dbx uses if you specify that name (without fully qualifying it) as the target of a debugging command. For example:

```
(dbx) func
wedge::wedge(char*, int, int, const point&, load_bearing_block*)
(dbx) which draw
`block_draw.cc`wedge::draw(unsigned long)
```

If a specified symbol name is not in a local scope, the which command searches for the first occurrence of the symbol along the *scope resolution search path*. If which finds the name, it reports the fully qualified name.

If at any place along the search path, the search finds multiple occurrences of *symbol* at the same scope level, dbx prints a message in the command pane reporting the ambiguity.

```
(dbx) which fid
More than one identifier 'fid'.
Select one of the following:
 0) Cancel
 1) `example'file1.c'fid
 2) 'example'file2.c'fid
```

dbx shows the overload display, listing the ambiguous symbols names. In the context of the which command, choosing from the list of occurrences does not affect the state of dbx or the program. Whichever occurrence you choose, dbx echoes the name.

The which command gives you a preview of what happens if you make *symbol* (in this example, block) an argument of a command that must operate on *symbol* (for example, a print command). In the case of ambiguous names, the overload display list indicates that dbx does not yet register which occurrence of two or more names it uses. dbx lists the possibilities and waits for you to choose one.

Viewing Variables, Members, Types, and Classes

The `whatis` command prints the declarations or definitions of identifiers, structs, types and C++ classes, or the type of an expression. The identifiers you can look up include variables, functions, fields, arrays, and enumeration constants.

For more information, see "whatis Command" and "whatis Command Used With C++" in the Using dbx Commands section of the Sun WorkShop online help.

Looking Up Definitions of Variables, Members, and Functions

To print out the declaration of an identifier, type:

```
(dbx) whatis identifier
```

Qualify the identifier name with file and function information as needed.

For C++, `whatis` *identifier* lists function template instantiations. Template definitions are displayed with `whatis -t` *identifier*. See "Looking Up Definitions of Types and Classes" on page 42.

To print out the member function, type:

```
(dbx) whatis block::draw
void block::draw(unsigned long pw);
(dbx) whatis table::draw
void table::draw(unsigned long pw);
(dbx) whatis block::pos
class point *block::pos();
Notice that table::pos is inherited from block:
(dbx) whatis table::pos
class point *block::pos();
```

To print out the data member, type:

```
(dbx) whatis block::movable
int movable;
```

On a variable, `whatis` tells you the variable's type.

```
(dbx) whatis the_table
class table *the_table;
```

On a field, `whatis` gives the field's type.

```
(dbx) whatis the_table->draw
void table::draw(unsigned long pw);
```

When you are stopped in a member function, you can look up the `this` pointer. In this example, the output from the `whatis` shows that the compiler automatically allocated this variable to a register.

```
(dbx) stop in brick::draw
(dbx) cont
// expose the blocks window (if exposed, hide then expose) to
force program to hit the breakpoint.
(dbx) where 1
brick::draw(this = 0x48870, pw = 374752), line 124 in
     "block_draw.cc"
(dbx) whatis this
class brick *this;
```

Looking Up Definitions of Types and Classes

The `-t` option of the `whatis` command displays a type. For C++, the list displayed by `whatis -t` includes template definitions and class template instantiations.

To print the declaration of a type or C++ class, type:

```
(dbx) whatis -t type_or_class_name
```

To see inherited members, the `whatis` command takes an `-r` option (for recursive) that displays the declaration of a specified class together with the members it inherits from parent classes.

```
(dbx) whatis -t -r class_name
```

The output from a `whatis -r` query may be long, depending on the class hierarchy and the size of the classes. The output begins with the list of members inherited from the most ancestral class. The inserted comment lines separate the list of members into their respective parent classes.

Here are two examples, using the class `table`, a child class of the parent class `load_bearing_block`, which is, in turn, a child class of `block`.

Without `-r`, `whatis` reports the members declared in class `table`:

```
(dbx) whatis -t class table
class table : public load_bearing_block {
public:
    table::table(char *name, int w, int h, const class point
&pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

Here are results when `whatis -r` is used on a child class to see members it inherits:

```
(dbx) whatis -t -r class table
class table : public load_bearing_block {
public:
  /* from base class table::load_bearing_block::block */
  block::block();
  block::block(char *name, int w, int h, const class point &pos,
class load_bearing_block *blk);
    virtual char *block::type();
    char *block::name();
    int block::is_movable();
//  deleted several members from example protected:
    char *nm;
    int movable;
    int width;
    int height;
    class point  position;
    class load_bearing_block *supported_by;
    Panel_item panel_item;
    /* from base class table::load_bearing_block */
public:
    load_bearing_block::load_bearing_block();
    load_bearing_block::load_bearing_block(char *name, int w, int
h,const class point &pos, class load_bearing_block *blk);
    virtual int load_bearing_block::is_load_bearing();
    virtual class list *load_bearing_block::supported_blocks();
    void load_bearing_block::add_supported_block(class block &b);
    void load_bearing_block::remove_supported_block(class block
&b);
```

```
    virtual void load_bearing_block::print_supported_blocks();
    virtual void load_bearing_block::clear_top();
    virtual void load_bearing_block::put_on(class block &object);
    class point load_bearing_block::get_space(class block
&object);
    class point load_bearing_block::find_space(class block
&object);
    class point load_bearing_block::make_space(class block
&object);
protected:
    class list *support_for;
    /* from class table */
public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

Using the Auto-Read Facility

In general, compile the entire program you want to debug using the `-g` option. Depending on how the program was compiled, the debugging information generated for each program and shared library module is stored in either the object code file (`.o` file) for each program and shared library module, or the program executable file.

When you compile with the `-g -c` compiler option, debugging information for each module remains stored in its `.o` file. `dbx` then reads in debugging information for each module automatically, as it is needed, during a session. This read-on-demand facility is called *Auto-Read*. Auto-Read is the default for `dbx`.

Auto-Read saves considerable time when yau are loading a large program into `dbx`. Auto-Read depends on the continued presence of the program `.o` files in a location known to `dbx`.

Note – If you archive `.o` files into `.a` files, and link using the archive libraries, you can then remove the associated `.o` files, but you must keep the `.a` files.

By default, dbx looks for files in the directory where they were when the program was compiled and the .o files in the location from which they were linked, using the absolute path recorded at compile time. If the files are not there, use the pathmap command to set the search path.

If no object file is produced, debugging information is stored in the executable. That is, for a compilation that does not produce .o files, the compiler stores all the debugging information in the executable. The debugging information is read the same way as for applications compiled with the -xs option. See "Debugging Without the Presence of .o Files" next.

Debugging Without the Presence of .o Files

Programs compiled with -g -c store debugging information for each module in the module's .o file. Auto-Read requires the continued presence of the program and shared library .o files.

When it is not feasible to keep program .o files or shared library .o files for modules that you want to debug, compile the program using the compiler -xs option (in addition to -g). You can have some modules compiled with -xs and some without. The -xs option instructs the compiler to have the linker place all the debugging information in the program executable; therefore the .o files do not have to be present to debug those modules.

In dbx 4.0, the debugging information for modules compiled with the -xs option is loaded during dbx startup. For a large program compiled with -xs, this might cause dbx to start slowly.

In dbx 5.0, the loading of debugging information for modules compiled with -xs is also delayed in the same way as the debugging information stored in .o files. However, you can instruct dbx to load the debugging information for modules compiled with -xs during startup. The dbx environment variable delay_xs lets you turn off the delayed loading of debugging information for modules compiled with -xs. To set the environment variable, add this line to your .dbxrc file:

```
dbxenv delay_xs off
```

You can also set this variable using the Debugging Options dialog box in Sun WorkShop Debugging. See "Delaying Loading of Modules Compiled with -xs" in the Using the Debugging Window section of the Sun WorkShop online help.

Listing Debugging Information for Modules

The `module` command and its options help you to keep track of program modules during the course of a debugging session. Use the `module` command to read in debugging information for one or all modules. Normally, `dbx` automatically and "lazily" reads in debugging information for modules as needed.

To read in debugging information for a module *name*, type:

```
(dbx) module [-f] [-q] name
```

To read in debugging information for all modules, type:

```
(dbx) module [-f] [-q] -a
```

where:

`-f`	Forces reading of debugging information, even if the file is newer than the executable.
`-q`	Specifies quiet mode.
`-v`	Specifies verbose mode, which prints language, file names, and so on.

To print the name of the current module, type:

```
(dbx) module
```

Listing Modules

The `modules` command helps you keep track of modules by listing module names.

To list the names of modules containing debugging information that have already been read into `dbx`, type:

```
(dbx) modules [-v] -read
```

To list names of all program modules (whether or not they contain debugging information), type:

```
(dbx) modules [-v]
```

To list all program modules that contain debugging information, type:

```
(dbx) modules [-v] -debug
```

where:

-v Specifies verbose mode, which prints language, file names, and so on.

CHAPTER 4

Controlling Program Execution

The commands used for running, stepping, and continuing (`run`, `rerun`, `next`, `step`, and `cont`) are called *process control* commands. Used together with the event management commands described in Chapter 6, you can control the run-time behavior of a program as it executes under `dbx`.

This chapter is organized into the following sections:

- Running a Program
- Attaching `dbx` to a Running Process
- Detaching `dbx` From a Process
- Stepping Through a Program
- Using Ctrl+C to Stop a Process

Running a Program

When you first load a program into `dbx`, `dbx` visits the program's "main" block (`main` for C, C++, and Fortran 90; `MAIN` for FORTRAN 77). `dbx` waits for you to issue further commands; you can visit code or use event management commands.

You can set breakpoints in the program before running it. Use the `run` command to start program execution.

To run a program in `dbx` without arguments, type:

```
(dbx) run
```

You can optionally add command-line arguments and redirection of input and output.

```
(dbx) run [arguments][ < input_file] [ > output_file]
```

Output from the `run` command overwrites an existing file even if you have set `noclobber` for the shell in which you are running `dbx`.

The `run` command without arguments restarts the program using the previous arguments and redirection. The `rerun` command restarts the program and clears the original arguments and redirection. For more information, see "rerun Command" in the Using dbx Commands section of the Sun WorkShop online help.

Attaching dbx to a Running Process

You might need to debug a program that is already running. You would attach to a running process if:

- You wanted to debug a running server, and you did not want to stop or kill it.
- You wanted to debug a running program that has a graphical user interface, and you didn't want to restart it.
- Your program was looping indefinitely, and you want to debug it without killing it.

You can attach `dbx` to a running program by using the program's *pid* number as an argument to the `dbx debug` command.

Once you have debugged the program, you can then use the `detach` command to take the program out from the control of `dbx` without terminating the process.

If you quit `dbx` after attaching it to a running process, `dbx` implicitly detaches before terminating.

To attach `dbx` to a program that is running independently of `dbx`, you can use either the `attach` command or the `debug` command.

To attach `dbx` to a process that is already running, type:

```
(dbx) debug program_name process_ID

or

(dbx) attach program_name process_ID
```

You can substitute a - (dash) for the *program_name*; `dbx` automatically finds the program associated with the *process_ID* and loads it.

For more information, see "debug Command" and "attach Command" in the Using dbx Commands section of the Sun WorkShop online help.

If dbx is not running, start dbx by typing:

```
% dbx program_name process_ID
```

After you have attached dbx to a program, the program stops executing. You can examine it as you would any program loaded into dbx. You can use any event management or process control command to debug it.

You can also attach dbx to a running process in the Sun WorkShop Debugging window by choosing Debug ➤ Attach Process. For more information, see "Attaching to a Running Process" in the Using the Debugging Window section of the Sun WorkShop online help.

Detaching dbx From a Process

When you have finished debugging the program, use the detach command to detach dbx from the program. The program then resumes running independently of dbx.

To detach a process from running under the control of dbx:

```
(dbx) detach
```

For more information, see "detach Command" in the Using dbx Commands section of the Sun WorkShop online help.

You can also detach dbx from a process in the Sun WorkShop Debugging window by choosing Execute ➤ Detach Process. For more information, see "Detaching From a Process" in the Using the Debugging Window section of the Sun WorkShop online help.

Stepping Through a Program

dbx supports two basic single-step commands: next and step, plus a variant of step, called step up. Both the next command and the step command let the program execute one source line before stopping again.

If the line executed contains a function call, the `next` command allows the call to be executed and stops at the following line ("steps over" the call). The `step` command stops at the first line in a called function ("steps into" the call). You can also step over a function in the Sun WorkShop Debugging window by choosing Execute ➤ Step Over or clicking the Step Over button on the tool bar. You can step into a function by choosing Execute ➤ Step Into or clicking the Step Into button on the tool bar.

The `step up` command returns the program to the caller function after you have stepped into a function. You can also step out of a function in the Sun WorkShop Debugging window by choosing Execute ➤ Step Out or clicking the Step Out button on the tool bar.

For more information, see "Program Stepping" in the Using the Debugging Window section of the Sun WorkShop online help.

Single Stepping

To single step a specified number of lines of code, use the `dbx` commands `next` or `step` followed by the number of lines [*n*] of code you want executed.

```
(dbx) next n

or

(dbx) step n
```

The `step_granularity` environment variable determines the granularity of source line stepping; that is, the number of `next` commands needed to step through a line of code. For more information, see "step_granularity Environment Variable" in the Using dbx Commands section of the Sun WorkShop online help.

For more information on the commands, see "next Command" and "step Command" in the Using dbx Commands section of the Sun WorkShop online help.

Continuing Execution of a Program

To continue a program, use the `cont` command.

```
(dbx) cont
```

You can also continue execution of a program in the Sun WorkShop Debugging window by choosing Execute ➤ Continue or clicking the Continue button on the tool bar.

The `cont` command has a variant, `cont at` *line_number*, which lets you specify a line other than the current program location line at which to resume program execution. This allows you to skip over one or more lines of code that you know are causing problems, without having to recompile.

To continue a program at a specified line, type:

```
(dbx) cont at 124
```

The line number is evaluated relative to the file in which the program is stopped; the line number given must be within the scope of the current function.

Using `cont at` *line_number* with `assign`, you can avoid executing a line of code that contains a call to a function that might be incorrectly computing the value of some variable.

To resume program execution at a specific line:

1. **Use `assign` to give the variable a correct value.**
2. **Use `cont at` *line_number* to skip the line that contains the function call that would have computed the value incorrectly.**

   Assume that a program is stopped at line 123. Line 123 calls a function, `how_fast()`, that computes incorrectly a variable, `speed`. You know what the value of speed should be, so you assign a value to `speed`. Then you continue program execution at line 124, skipping the call to `how_fast()`.

```
(dbx) assign speed = 180; cont at 124;
```

For more information, see "cont Command" in the Using dbx Commands section of the Sun WorkShop online help.

If you use the `cont` command with a `when` breakpoint command, the program skips the call to `how_fast()` each time the program attempts to execute line 123.

```
(dbx) when at 123 { assign speed = 180; cont at 124;}
```

For more information on the `when` command, see:

- "Setting a `stop` Breakpoint at a Line of Source Code" on page 58
- "Setting Breakpoints in Member Functions of Different Classes" on page 61

- "Setting Breakpoints in Member Functions of the Same Class" on page 61
- "Setting Multiple Breakpoints in Nonmember Functions" on page 61
- "when Command" in the Using dbx Commands section of the Sun WorkShop online help

Calling a Function

When a program is stopped, you can call a function using the `dbx call` command, which accepts values for the parameters that must be passed to the called function.

To call a procedure, type the name of the function and supply its parameters. For example:

```
(dbx) call change_glyph(1,3)
```

While the parameters are optional, you must type the parentheses after the *function_name*. For example:

```
(dbx) call type_vehicle()
```

You can call a function explicitly, using the `call` command, or implicitly, by evaluating an expression containing function calls or using a conditional modifier such as `stop in glyph -if animate()`.

A C++ virtual function can be called like any other function using the `print` command or `call` command (see "print Command" or "call Command" in the Using dbx Commands section of the Sun WorkShop online help), or any other command that executes a function call.

If the source file in which the function is defined was compiled with the `-g` option, or if the prototype declaration is visible at the current scope, `dbx` checks the number and type of arguments and issues an error message if there is a mismatch. Otherwise, `dbx` does not check the number of parameters and proceeds with the call.

By default, after every `call` command, `dbx` automatically calls `fflush(stdout)` to ensure that any information stored in the I/O buffer is printed. To turn off automatic flushing, set the `dbx` environment variable `output_autoflush` to `off`.

For C++, `dbx` handles the implicit `this` pointer, default arguments, and function overloading. The C++ overloaded functions are resolved automatically if possible. If any ambiguity remains (for example, functions not compiled with `-g`), `dbx` displays a list of the overloaded names.

When you use the `call` command, `dbx` behaves as though you used the `next` command, returning from the called function. However, if the program encounters a breakpoint in the called function, `dbx` stops the program at the breakpoint and issues a message. If you now type a `where` command, the stack trace shows that the call originated from `dbx` command level.

If you continue execution, the call returns normally. If you attempt to kill, run, rerun, or debug, the command aborts as `dbx` tries to recover from the nesting. You can then re-issue the command. Alternatively, you can use the command `pop -c` to pop all frames up to the most recent call.

Using Ctrl+C to Stop a Process

You can stop a process running in `dbx` by pressing Ctrl+C (`^C`). When you stop a process using `^C`, `dbx` ignores the `^C`, but the child process accepts it as a `SIGINT` and stops. You can then inspect the process as if it had been stopped by a breakpoint.

To resume execution after stopping a program with `^C`, use the `cont` command. You do not need to use the `cont` optional modifier, `sig` *signal_name*, to resume execution. The `cont` command resumes the child process after cancelling the pending signal.

CHAPTER 5

Setting Breakpoints and Traces

A breakpoint is a location where an action occurs, at which point the program stops executing. You can set a trace that displays information about an event in your program, such as a change in the value of a variable. Although a trace's behavior is different from that of a breakpoint, traces and breakpoints share similar event handlers.

This chapter describes how to set, clear, and list breakpoints and traces, and how to use watchpoints.

The chapter is organized into the following sections:

- Setting Breakpoints
- Tracing Code
- Listing and Clearing Event Handlers
- Setting Breakpoint Filters
- Efficiency Considerations

Setting Breakpoints

In `dbx`, you can use three types of breakpoint action commands to set source-level breakpoints:

- `stop` breakpoints–If the program arrives at a breakpoint created with a `stop` command, the program halts. The program cannot resume until you issue another debugging command, such as `cont`, `step`, or `next`.
- `when` breakpoints–The program halts and `dbx` executes one or more debugging commands, then the program continues (unless one of the commands is `stop`).
- `trace` breakpoints–The program halts and an event-specific `trace` information line is emitted, then the program continues.

To set machine-level breakpoints, use the `stopi`, `wheni`, and `tracei` commands (see Chapter 17).

Setting a `stop` Breakpoint at a Line of Source Code

You can set a breakpoint at a line number, using the `dbx stop at` command, where *n* is a source code line number and *filename* is an optional program file name qualifier.

```
(dbx) stop at filename: n
```

For example:

```
(dbx) stop at main.cc:3
```

If the line specified in a `stop` or `when` command is not an executable line of source code, `dbx` sets the breakpoint at the next executable line. If there is no executable line, `dbx` issues an error.

You can also set a breakpoint at a location in the Sun WorkShop Breakpoints window. For more information, see "Breaking at a Location" in the Using the Debugging Window section of the Sun WorkShop online help.

Setting a `stop` Breakpoint in a Function

You can set a breakpoint in a function, using the `dbx stop in` command:

```
(dbx) stop in function
```

An In Function breakpoint suspends program execution at the beginning of the first source line in a procedure or function.

You can also set a breakpoint in a function in the Sun WorkShop Breakpoints window. For more information, see "Breaking in a Function" in the Using the Debugging Window section of the Sun WorkShop online help.

dbx should be able to determine which variable or function you are referring to except when:

- You reference an overloaded function by name only.
- You reference a function or variable with a leading ''.

Consider the following example:

```
int foo(double);
             int foo(int);
             int bar();
             class x {
                int bar();
};
```

When you stop at a non-member function, you can type:

```
stop in foo(int)
```

to set a breakpoint at the global `foo(int)`.

To set a breakpoint at the member function you can use the command:

```
stop in x::bar()
```

If you type:

```
stop in foo
```

`dbx` cannot determin whether you mean the global function `foo(int)` or the global function `foo(double)` and may be forced to display an overloaded menu for clarification.

If you type:

```
stop in ''bar
```

`dbx` cannot determine whether you mean the global function `bar()` or the member function `bar()` and display an overloaded menu.

Setting a when Breakpoint at a Line

A when breakpoint command accepts other dbx commands such as list, letting you write your own version of trace.

```
(dbx) when at 123 { list $lineno;}
```

The when command operates with an implied cont command. In the example above, after listing the source code at the current line, the program continues executing.

Setting a Breakpoint in a Dynamically Linked Library

dbx provides full debugging support for code that uses the programmatic interface to the run-time linker: code that calls dlopen(), dlclose() and their associated functions. The run-time linker binds and unbinds shared libraries during program execution. Debugging support for dlopen()/dlclose() lets you step into a function or set a breakpoint in functions in a dynamically shared library just as you can in a library linked when the program is started.

There are three exceptions:

- You cannot set a breakpoint in a library loaded by dlopen() before that library is loaded by dlopen().
- You cannot set a breakpoint in a filter library loaded by dlopen() until the first function in it is called.
- When a library is loaded by dlopen(), an initialization routine named _init() is called. This routine may call other routines in the library. dbx cannot place breakpoints in the loaded library until after this initialization is completed. You cannot have dbx stop at _init() in a library loaded by dlopen().

Setting Multiple Breaks in C++ Programs

You can check for problems related to calls to members of different classes, calls to any members of a given class, or calls to overloaded top-level functions. You can use a keyword—inmember, inclass, infunction, or inobject—with a stop, when, or trace command to set multiple breaks in C++ code.

Setting Breakpoints in Member Functions of Different Classes

To set a breakpoint in each of the object-specific variants of a particular member function (same member function name, different classes), use `stop inmember`.

To set a `when` breakpoint, use `when inmember`.

For example, if the function `draw` is defined in several different classes, then to place a breakpoint in each function, type:

```
(dbx) stop inmember draw
```

You can also set an In Member breakpoint in the Sun WorkShop Breakpoints window. For more information, see "Setting an In Member Breakpoint" in the Using the Debugging Window section of the Sun WorkShop online help.

Setting Breakpoints in Member Functions of the Same Class

To set a breakpoint in all member functions of a specific class, use the `stop inclass` command.

To set a `when` breakpoint, use `when inclass`.

To set a breakpoint in all member functions of the class `draw`, type:

```
(dbx) stop inclass draw
```

You can also set an In Class breakpoint in the Sun WorkShop Breakpoints window. For more information, see "Setting an In Class Breakpoint" in the Using the Debugging Window section of the Sun WorkShop online help.

Breakpoints are inserted only in the class member functions defined in the class, not those that it might inherit from base classes.

Due to the large number of breakpoints that may be inserted by `stop inclass` and other breakpoint selections, you should be sure to set the `dbx` environment variable `step_events` to `on` to speed up the `step` and `next` commands (see "Efficiency Considerations" on page 68).

Setting Multiple Breakpoints in Nonmember Functions

To set multiple breakpoints in nonmember functions with overloaded names (same name, different type or number of arguments), use the `stop infunction` command.

To set a `when` breakpoint, use `when infunction`.

For example, if a C++ program has defined two versions of a function named `sort()`(one that passes an `int` type argument and the other a `float`) then, to place a breakpoint in both functions, type:

```
(dbx) when infunction sort {cmd;}
```

You can also set an In Function breakpoint in the Sun WorkShop Breakpoints window. For more information, see "Breaking in a Function" in the Using the Debugging Window section of the Sun WorkShop online help.

Setting Breakpoints in Objects

Set an In Object breakpoint to check the operations applied to a specific object. An In Object breakpoint suspends program execution in all nonstatic member functions of the object's class when called from the object.

To set a breakpoint in object `foo`, type:

```
(dbx) stop inobject &foo
```

You can also set an In Object breakpoint in the Sun WorkShop Breakpoints window. For more information, see "Setting an In Object Breakpoint" in the Using the Debugging Window section of the Sun WorkShop online help.

Tracing Code

Tracing displays information about the line of code about to be executed, a function about to be called, or a variable about to be changed. For more information, see "Code Tracing" in the Using the Debugging Window section of the Sun WorkShop online help.

Setting a Trace

Set a trace by typing a `trace` command at the command line. The basic syntax of the `trace` command is:

```
trace event-specification [ modifier ]
```

For the complete syntax of the trace command, see "trace Command" in the Using dbx Commands section of the Sun WorkShop online help.

You can also set a trace in the Sun WorkShop Breakpoints window. For more information, see "Setting Up a Trace" in the Using the Debugging Window section of the Sun WorkShop online help.

The information a trace provides depends on the type of *event* associated with it (see "Setting Event Specifications" on page 72.

Controlling the Speed of a Trace

In many programs, code execution is too fast for you to view the code. The `dbx` environment variable `trace_speed` lets you control the delay after each trace is printed. The default delay is `0.5` seconds.

To set the interval between execution of each line of code during a trace, type:

```
dbxenv trace_speed number
```

You can also set the speed of a trace in the Sun WorkShop Debugging Options dialog box. For more information, see "Setting the Trace Speed" in the Using the Debugging Window section of the Sun WorkShop online help.

Listing and Clearing Event Handlers

Often, you set more than one breakpoint or trace handler during a debugging session. `dbx` supports commands for listing and clearing them.

Listing Breakpoints and Traces

To display a list of all active breakpoints, use the `status` command to display ID numbers in parentheses, which can then be used by other commands.

`dbx` reports multiple breakpoints set with the `inmember`, `inclass`, and `infunction` keywords as a single set of breakpoints with one status ID number.

Deleting Specific Breakpoints Using Handler ID Numbers

When you list breakpoints using the `status` command, `dbx` displays the ID number assigned to each breakpoint when it was created. Using the `delete` command, you can remove breakpoints by ID number, or use the keyword `all` to remove all breakpoints currently set anywhere in the program.

To delete breakpoints by ID number, type:

```
(dbx) delete 3 5
```

To delete all breakpoints set in the program currently loaded in `dbx`, type:

```
(dbx) delete all
```

For more information, see “delete Command” in the Using dbx Commands section of the Sun WorkShop online help.

Watchpoints

Watchpointing is the capability of `dbx` to note when the value of a variable or expression has changed. You can set watchpoints using the `stop` command.

Stopping Execution When Modified

To stop program execution when the contents of an address is written to, type:

```
(dbx) stop modify &variable
```

Keep these points in mind when using the `stop modify` command:

- The event occurs when a variable is written to even if it is the same value.
- The event occurs *before* the instruction that wrote to the variable is executed, although the new contents of the memory are preset by `dbx` by emulating the instruction.
- You cannot use addresses of stack variables, for example, auto function local variables.

You can also set an On Modify breakpoint in the Sun WorkShop Breakpoints window. For more information, see "Setting a Custom On Value Change Breakpoint" in the Using the Debugging Window section of the Sun WorkShop online help.

Stopping Execution When Variables Change

To stop program execution if the value of a specified variable has changed, type:

```
(dbx) stop change variable
```

Keep these points in mind when using the `stop change` command:

- `dbx` stops the program at the line *after* the line that caused a change in the value of the specified variable.
- If *variable* is local to a function, the variable is considered to have changed when the function is first entered and storage for *variable* is allocated. The same is true with respect to parameters.

You can also set an On Value Change breakpoint in the Sun WorkShop Breakpoints window. For more information, see "Breaking On Value Change" in the Using the Debugging Window section of the Sun WorkShop online help.

`dbx` implements `stop change` by causing automatic single stepping together with a check on the value at each step. Stepping skips over library calls if the library was not compiled with the `-g` option. So, if control flows in the following manner, `dbx` does not trace the nested `user_ routine2` because tracing skips the library call and the nested call to `user_routine2`.

```
user_routine calls
   library_routine, which calls
     user_routine2, which changes variable
```

The change in the value of *variable* appears to have occurred after the return from the library call, not in the middle of `user_routine2`.

`dbx` cannot set a breakpoint for a change in a block local variable—a variable nested in {}. If you try to set a breakpoint or trace in a block local "nested" variable, `dbx` issues an error informing you that it cannot perform this operation.

Stopping Execution on a Condition

To stop program execution if a conditional statement evaluates to true, type:

```
(dbx) stop cond condition
```

You can also set a custom On Condition breakpoint in the Sun WorkShop Breakpoints window. For more information, see "Setting a Custom On Condition Breakpoint" in the Using the Debugging Window section of the Sun WorkShop online help.

Faster `modify` Event

A faster way of setting watchpoints is to use the `modify` event. See "Predefined Events" in the Using dbx Commands section of the Sun WorkShop online help.

Instead of automatically single-stepping the program, the `modify` event uses a page protection scheme that is much faster. The speed depends on how many times the page on which the variable you are watching is modified, as well as the overall system call rate of the program being debugged.

Setting Breakpoint Filters

In `dbx`, most of the event management commands also support an optional *event filter* modifier statement. The simplest filter instructs `dbx` to test for a condition after the program arrives at a breakpoint or trace handler, or after a watch condition occurs. For information on event modifiers, see "Event Specification Modifiers" on page 81.

If this filter condition evaluates to true (non 0), the event command applies. If the condition evaluates to false (0), `dbx` continues program execution as if the event had never happened.

To set a breakpoint at a line or in a function that includes a conditional filter, add an optional `-if` *condition* modifier statement to the end of a `stop` or `trace` command.

The condition can be any valid expression, including function calls, returning Boolean or integer in the language current at the time the command is entered.

Note – With a location-based breakpoint like `in` or `at`, the scope is that of the breakpoint location. Otherwise, the scope of the condition is the scope at the time of entry, not at the time of the event. You might have to use syntax to specify the scope precisely.

For example, these two filters are not the same:

```
stop in foo -if a>5
stop cond a>5
```

The former breaks at `foo` and tests the condition. The latter automatically single-steps and tests for the condition.

New users sometimes confuse setting a conditional event command (a watch-type command) with using filters. Conceptually, "watching" creates a *precondition* that must be checked before each line of code executes (within the scope of the watch). But even a breakpoint command with a conditional trigger can also have a filter attached to it.

Consider this example:

```
(dbx) stop modify &speed -if speed==fast_enough
```

This command instructs `dbx` to monitor the variable, *speed*; if the variable *speed* is written to (the "watch" part), then the `-if` filter goes into effect. `dbx` checks whether the new value of *speed* is equal to `fast_enough`. If it is not, the program continues, "ignoring" the `stop`.

In `dbx` syntax, the filter is represented in the form of an `[-if` *condition*`]` statement at the end of the command.

```
stop in function [-if condition]
```

Efficiency Considerations

Various events have different degrees of overhead in respect to the execution time of the program being debugged. Some events, like the simplest breakpoints, have practically no overhead. Events based on a single breakpoint have minimal overhead.

Multiple breakpoints such as `inclass`, that might result in hundreds of breakpoints, have an overhead only during creation time. This is because `dbx` uses permanent breakpoints; the breakpoints are retained in the process at all times and are not taken out on every stoppage and put in on every `cont`.

Note – In the case of `step` and `next`, by default all breakpoints are taken out before the process is resumed and reinserted once the step completes. If you are using many breakpoints or multiple breakpoints on prolific classes, the speed of `step` and `next` slows down considerably. Use the `dbx step_events` environment variable to control whether breakpoints are taken out and reinserted after each `step` or `next`.

The slowest events are those that utilize automatic single stepping. This might be explicit and obvious as in the `trace step` command, which single steps through every source line. Other events, like the watchpoints `stop change` *expression* or `trace cond` *variable* not only single step automatically but also have to evaluate an expression or a variable at each step.

These events are very slow, but you can often overcome the slowness by bounding the event with a function using the `-in` modifier. For example:

```
trace next -in mumble
stop change clobbered_variable -in lookup
```

Do not use `trace -in main` because the `trace` is effective in the functions called by `main` as well. Do use it in the cases where you suspect that the `lookup()` function is clobbering your variable.

CHAPTER 6

Event Management

Event management refers to the capability of `dbx` to perform actions when events take place in the program being debugged.

This chapter is organized into the following sections:

- Event Handlers
- Creating Event Handlers
- Manipulating Event Handlers
- Using Event Counters
- Setting Event Specifications
- Parsing and Ambiguity
- Using Predefined Variables
- Setting Event Handler Examples

Event Handlers

Event management is based on the concept of a *handler*. The name comes from an analogy with hardware interrupt handlers. Each event management command typically creates a handler, which consists of an *event specification* and a series of side-effect actions. (See "Setting Event Specifications" on page 72.) The event specification specifies the event that will trigger the handler.

When the event occurs and the handler is triggered, the handler evaluates the event according to any modifiers included in the event specification. (See "Event Specification Modifiers" on page 81.) If the event meets the conditions imposed by the modifiers, the handler's side-effect actions are performed (that is, the handler "fires").

An example of the association of a program event with a `dbx` action is setting a breakpoint on a particular line.

The most generic form of creating a handler is by using the `when` command.

```
when event-specification {action; ... }
```

Examples in this chapter show how you can write a command (like `stop`, `step`, or `ignore`) in terms of `when`. These examples are meant to illustrate the flexibility of `when` and the underlying *handler* mechanism, but they are not always exact replacements.

Creating Event Handlers

Use the commands `when`, `stop`, and `trace` to create event handlers.

`stop` is shorthand for a common `when` idiom.

```
when event-specification { stop -update; whereami; }
```

An *event-specification* is used by the event management commands `stop`, `when`, and `trace` to specify an event of interest. (see "Setting Event Specifications" on page 72).

Most of the `trace` commands can be handcrafted using the `when` command, `ksh` functionality, and event variables. This is especially useful if you want stylized tracing output.

For detailed information, see "when Command," "stop Command," and "trace Command" in the Using dbx Commands section of the Sun WorkShop online help.

Every command returns a number known as a handler id (*hid*). You can access this number using the predefined variable `$newhandlerid`.

An attempt has been made to make the `stop` and `when` commands conform to the handler model. However, backward compatibility with previous `dbx` releases forces some deviations.

For example, the following samples from an earlier `dbx` release are equivalent.

Old	New
`when` *cond body*	`when step -if` *cond body*
`when` *cond* `in` *func body*	`when next -if` *cond* `-in` *func body*

These samples illustrate that *cond* is not a pure event; there is no internal handler for conditions.

Manipulating Event Handlers

You can use the following commands to manipulate event handlers. For more information on each command, see the cited topic in the Using dbx Commands section of the Sun WorkShop online help.

- `status` – lists handlers (see "status Command").
- `delete` – deletes all handlers including temporary handlers (see "delete Command").
- `clear` – deletes handlers based on breakpoint position (see "clear Command").
- `handler -enable` – enables handlers (see "handler Command").
- `handler -disable` – disables handlers.
- `cancel` – cancels signals and lets the process continue (see "cancel Command").

Using Event Counters

An event handler has a trip counter, which has a count limit. Whenever the specified event occurs, the counter is incremented. The action associated with the handler is performed only if the count reaches the limit, at which point the counter is automatically reset to 0. The default limit is 1. Whenever a process is rerun, all event counters are reset.

You can set the count limit using the `-count` modifier with a `stop`, `when`, or `trace` command (see "`-count n -count infinity`" on page 82). Otherwise, use the `handler` command to individually manipulate event handlers:.

```
handler [ -count | -reset ] hid new-count new-count-limit
```

Setting Event Specifications

Event specifications are used by the `stop, when`, and `trace` commands to denote event types and parameters. The format consists of a keyword representing the event type and optional parameters. For more information, see "Event Specification" in the Using dbx Commands section of the Sun WorkShop online help.

Breakpoint Event Specifications

The following are event specifications for breakpoint events.

`in` *function*

The function has been entered, and the first line is about to be executed. If the `-instr` modifier is used (see "`-instr`" on page 82), it is the first instruction of the function about to be executed. The *func* specification can take a formal parameter signature to help with overloaded function names or template instance specification. For example:

```
stop in mumble(int, float, struct Node *)
```

Note – Do not confuse `in` *function* with the `-in` *function* modifier.

`at` [*filename:*]*lineno*

The designated line is about to be executed. If you specify *filename*, then the designated line in the specified file is about to be executed. The file name can be the name of a source file or an object file. Although quotation marks are not required, they may be necessary if the file name contains special characters. If the designated line is in template code, a breakpoint is placed on all instances of that template.

`infunction` *function*

Equivalent to `in` *function* for all overloaded functions named *function* or all template instantiations thereof.

`inmember` *function*
`inmethod` *function*

Equivalent to `in` *function* for the member function named *function* for every class.

`inclass` *classname*

Equivalent to `in` *function* for all member functions that are members of *classname*.

`inobject` *object-expression*

A member function called on the specific object at the address denoted by *object-expression* has been called.

Watchpoint Event Specifications

The following are event specifications for watchpoint events. For more information, see "Watchpoint Specification" in the Using dbx Commands section of the Sun WorkShop online help.

`access` *mode addr-expression* [, *byte-size-expression*]

The memory specified by *addr-expression* has been accessed.

mode specifies that the memory was accessed. It can be composed of one or all of the letters:

`r`	The memory has been read.
`w`	The memory has been written to.
`x`	The memory has been executed.

mode can also contain either of the following:

`a`	Stops the process after the access (default).
`b`	Stops the process before the access.

In both cases the program counter will point at the offending instruction. The "before" and "after" refer to the side effect.

address-expression is any expression that can be evaluated to produce an address. If you give a symbolic expression, the size of the region to be watched is automatically deduced; you can override it by specifying *byte-size-expression*. You can also use nonsymbolic, typeless address expressions; in which case, the size is mandatory. For example:

```
stop access 0x5678, sizeof(Complex)
```

The `access` command has the limitation that no two matched regions may overlap.

Note – The `access` event specification is a replacement for the `modify` event specification. While both syntaxes work on Solaris 2.6, Solaris 7, and Solaris 8, on all of these operating environments except Solaris 2.6, `access` suffers the same limitations as `modify` and accepts only a mode of `wa`.

`change` *variable*

The value of *variable* has changed.

`cond` *condition-expression*

The condition denoted by *cond-expression* evaluates to true. You can specify any expression for *cond-expression*, but it must evaluate to an integral type.

`modify` *address-expression* `[,`*byte-size-expression*`]`

The specified address range has been modified. This is the older watchpoint facility.

address-expression is any expression that can be evaluated to produce an address. If you give a symbolic expression, the size of the region to be watched is automatically deduced; you can override it by specifying *byte-size-expression*. You can also use nonsymbolic, typeless address expressions; in which case, the size is mandatory. For example:

```
stop modify 0x5678, sizeof(Complex)
```

Note – Multithreaded applications are prone to deadlock so mt watchpoints are nominally disallowed. They can be turned on by setting the `dbx` environment variable `mt_watchpoints`.

System Event Specifications

The following are event specifications for system events.

`dlopen [` *lib-path* `] | dlclose [` *lib-path* `]`

These events occur after a `dlopen`() or a `dlclose`() call succeeds. A `dlopen`() or `dlclose`() call can cause more than one library to be loaded. The list of these libraries is always available in the predefined variable `$dllist`. The first word in `$dllist` is a "+" or a "-", indicating whether the list of libraries is being added or deleted.

lib-path is the name of a shared library. If it is specified, the event occurs only if the given library was loaded or unloaded. In that case, `$dlobj` contains the name of the library. `$dllist` is still available.

If *lib-path* begins with a `/`, a full string match is performed. Otherwise, only the tails of the paths are compared.

If *lib-path* is not specified, then the events always occur whenever there is any `dl`-activity. In this case, `$dlobj` is empty but `$dllist` is valid.

`fault` *fault*

The `fault` event occurs when the specified fault is encountered. The faults are architecture-dependent. The following set of faults known to `dbx` is defined in the `proc`(4) man page.

Fault	Description
`FLTILL`	Illegal instruction
`FLTPRIV`	Privileged instruction
`FLTBPT`	Breakpoint trap
`FLTTRACE`*	Trace trap (single step)
`FLTWATCH`	Watchpoint trap

Fault	Description
FLTACCESS*	Memory access (such as alignment)
FLTBOUNDS*	Memory bounds (invalid address)
FLTIOVF	Integer overflow
FLTIZDIV	Integer zero divide
FLTPE	Floating-point exception
FLTSTACK	Irrecoverable stack fault
FLTPAGE	Recoverable page fault

Note – BPT, TRACE, and BOUNDS are used by dbx to implement breakpoints, single-stepping, and watchpoints. Handling them might interfere with how dbx works.

These faults are taken from /sys/fault.h. *fault* can be any of those listed above, in uppercase or lowercase, with or without the FLT- prefix, or the actual numerical code.

lwp_exit

The lwp_exit event occurs when lwp has been exited. $lwp contains the id of the exited LWP (lightweight process).

sig *sig*

The sig *sig* event occurs when the signal is first delivered to the program being debugged. *sig* can be either a decimal number or the signal name in uppercase or lowercase; the prefix is optional. This is completely independent of the catch/ignore commands, although the catch command can be implemented as follows:

```
function simple_catch {
    when sig $1 {
        stop;
        echo Stopped due to $sigstr $sig
        whereami
    }
}
```

Note – When the `sig` event is received, the process has not seen it yet. Only if you continue the process with the specified signal is the signal forwarded to it.

`sig` *sig sub-code*

When the specified signal with the specified *sub-code* is first delivered to the child, the `sig` *sig sub-code* event occurs. As with signals, you can type the *sub-code* as a decimal number, in uppercase or lowercase; the prefix is optional.

`sysin` *code* | *name*

The specified system call has just been initiated, and the process has entered kernel mode.

The concept of system call supported by `dbx` is that provided by traps into the kernel as enumerated in `/usr/include/sys/syscall.h`.

This is not the same as the ABI notion of system calls. Some ABI system calls are partially implemented in user mode and use non-ABI kernel traps. However, most of the generic system calls (the main exception being signal handling) are the same between `syscall.h` and the ABI.

`sysout` *code* | *name*

The specified system call is finished, and the process is about to return to user mode.

`sysin | sysout`

Without arguments, all system calls are traced. Certain `dbx` features, for example, the `modify` event and runtime checking, cause the child to execute system calls for its own purposes and show up if traced.

Execution Progress Event Specifications

The following are event specifications for events pertaining to execution progress.

next

The `next` event is similar to the `step` event except that functions are not stepped into.

returns

The `returns` event is a breakpoint at the return point of the current *visited* function. The visited function is used so that you can use the `returns` event specification after giving a number of `step up` commands. The `returns` event is always `-temp` and can only be created in the presence of a live process.

returns *func*

The `returns` *func* event executes each time the given function returns to its call site. This is not a temporary event. The return value is not provided, but you can find integral return values by accessing the following registers:

Sparc	`$o0`
Intel	`$eax`

The event is another way of saying:

```
when in func { stop returns; }
```

step

The `step` event occurs when the first instruction of a source line is executed. For example, you can get simple tracing with:

```
when step { echo $lineno: $line; }
```

When enabling a step event, you instruct `dbx` to single-step automatically next time the `cont` command is used. You can implement the `step` command as follows:

```
alias step="when step -temp { whereami; stop; }; cont"
```

Other Event Specifications

The following are event specifications for other types of events.

attach

`dbx` has successfully attached to a process.

detach

dbx has successfully detached from the program being debugged.

lastrites

The process being debugged is about to expire. This can happen for the following reasons:

- The `_exit`(2) system call has been called. (This happens either through an explicit call or when `main`() returns.)
- A terminating signal is about to be delivered.
- The process is being killed by the `kill` command.

proc_gone

The `proc_gone` event occurs when `dbx` is no longer associated with a debugged process. The predefined variable `$reason` may be `signal`, `exit`, `kill`, or `detach`.

prog_new

The `prog_new` event occurs when a new program has been loaded as a result of `follow exec`.

Note – Handlers for this event are always permanent.

---

stop

The process has stopped. The `stop` event occurs whenever the process stops such that the user receives a prompt, particularly in response to a `stop` handler. For example, the following commands are equivalent:

```
display x
when stop {print x;}
```

sync

The process being debugged has just been executed with `exec`(). All memory specified in `a.out` is valid and present, but preloaded shared libraries have not been loaded. For example, `printf`, although available to `dbx`, has not been mapped into memory.

A `stop` on this event is ineffective; however, you can use the `sync` event with the `when` command.

syncrtld

The `syncrtld` event occurs after a `sync` (or `attach` if the process being debugged has not yet processed shared libraries). It executes after the dynamic linker startup code has executed and the symbol tables of all preloaded shared libraries have been loaded, but before any code in the `.init` section has run.

A `stop` on this event is ineffective; however, you can use the `syncrtld` event with the `when` command.

throw

The `throw` event occurs whenever any exception that is not unhandled or unexpected is thrown by the application.

throw *type*

If an exception *type* is specified with the `throw` event, only exceptions of that type cause the `throw` event to occur.

`throw -unhandled`

`-unhandled` is a special exception type signifying an exception that is thrown but for which there is no handler.

`throw -unexpected`

`-unexpected` is a special exception type signifying an exception that does not satisfy the exception specification of the function that threw it.

`timer` *seconds*

The `timer` event occurs when the program being debugged has been running for *seconds*. The timer used with this event is shared with `collector` command. The resolution is in milliseconds, so a floating point value for seconds is acceptable.

Event Specification Modifiers

An event specification modifier sets additional attributes of a handler, the most common kind being event filters. Modifiers must appear after the keyword portion of an event specification. A modifier begins with a dash (`-`). The following are the valid event specification modifiers.

`-if` *condition*

The condition is evaluated when the event specified by the event specification occurs. The side effect of the handler is allowed only if the condition evaluates to nonzero.

If the `-if` modifier is used with an event that has an associated singular source location, such as `in` or `at`, *condition* is evaluated in the scope corresponding to that location. Otherwise, qualify it with the desired scope.

`-in` *function*

The handler is active only while within the given function or any function called from *function*. The number of times the function is entered is reference counted to properly deal with recursion.

-disable

The `-disable` modifier creates the handler in the disabled state.

-count *n*
-count infinity

The `-count` *n* and `-count infinity` modifiers have the handler count from 0 (see "Using Event Counters" on page 71). Each time the event occurs, the count is incremented until it reaches *n*. Once that happens, the handler fires and the counter is reset to zero.

Counts of all enabled handlers are reset when a program is run or rerun. More specifically, they are reset when the `sync` event occurs.

-temp

Creates a temporary handler. Once the event has occurred it is automatically deleted. By default, handlers are not temporary. If the handler is a counting handler, it is automatically deleted only when the count reaches 0 (zero).

Use the `delete -temp` command to delete all temporary handlers.

-instr

Makes the handler act at an instruction level. This event replaces the traditional '`i`' suffix of most commands. It usually modifies two aspects of the event handler:

- Any message prints assembly-level rather than source-level information.
- The granularity of the event becomes instruction level. For instance, `step -instr` implies instruction-level stepping.

-thread *thread_ID*

The action is executed only if the thread that caused the event matches *thread_ID*.

-lwp *lwp_ID*

The action is executed only if the thread that caused the event matches *lwp_ID*.

-hidden

Hides the handler in a regular `status` command. Use `status -h` to see hidden handlers.

-perm

Normally all handlers are thrown away when a new program is loaded. Using the `-perm` modifier retains the handler across debuggings. A plain `delete` command does not delete a permanent handler. Use `delete -p` to delete a permanent handler.

Parsing and Ambiguity

The syntax for event specifications and modifiers is:

- Keyword driven
- Based on `ksh` conventions; everything is split into words delimited by spaces

Expressions can have spaces embedded in them, causing ambiguous situations. For example, consider the following two commands:

```
when a -temp
when a-temp
```

In the first example, even though the application might have a variable named *temp*, the `dbx` parser resolves the event specification in favor of `-temp` being a modifier. In the second example, `a-temp` is collectively passed to a language-specific expression parser. There must be variables named *a* and *temp* or an error occurs. Use parentheses to force parsing.

Using Predefined Variables

Certain read-only `ksh` predefined variables are provided. The following variables are always valid:

Variable	Definition
`$ins`	Disassembly of the current instruction.
`$lineno`	Current line number in decimal.
`$vlineno`	Current “visiting” line number in decimal.
`$line`	Contents of the current line.
`$func`	Name of the current function.
`$vfunc`	Name of the current “visiting” function.
`$class`	Name of the class to which `$func` belongs.
`$vclass`	Name of the class to which `$vfunc` belongs.
`$file`	Name of the current file.
`$vfile`	Name of the current file being visited.
`$loadobj`	Name of the current loadable object.
`$vloadobj`	Name of the current loadable object being visited.
`$scope`	Scope of the current PC in back-quote notation.
`$vscope`	Scope of the visited PC in back-quote notation.
`$funcaddr`	Address of `$func` in hex.
`$caller`	Name of the function calling `$func.`
`$dllist`	After a `dlopen` or `dlclose` event, contains the list of load objects just loaded or unloaded. The first word of `dllist` is a “+” or a “-” depending on whether a `dlopen` or a `dlclose` has occurred.
`$newhandlerid`	ID of the most recently created handler
$firedhandlers	List of handler ids that caused the most recent stoppage. The handlers on the list are marked with “*” in the output of the `status` command.
`$proc`	Process ID of the current process being debugged.
`$lwp`	Lwp ID of the current LWP.
`$thread`	Thread ID of the current thread.

Variable	Definition
`$prog`	Full path name of the program being debugged.
`$oprog`	Old, or original value of `$prog`. This is used to get back to what you were debugging following an `exec()`.
`$exitcode`	Exit status from the last run of the program. The value is an empty string if the process has not exited.

As an example, consider that `whereami` can be implemented as:

```
function whereami {
  echo Stopped in $func at line $lineno in file $(basename $file)
  echo "$lineno\t$line"
}
```

Variables Valid for `when` Command

The following variables are valid only within the body of a `when` command.

`$handlerid`

During the execution of the body, `$handlerid` is the id of the `when` command to which the body belongs. These commands are equivalent:

```
when X -temp { do_stuff; }
when X  { do_stuff; delete $handlerid; }
```

`$booting`

`$booting` is set to `true` if the event occurs during the *boot* process. Whenever a new program is debugged, it is first run without the user's knowledge so that the list and location of shared libraries can be ascertained. The process is then killed. This sequence is termed booting.

While booting is occurring, all events are still available. Use this variable to distinguish the `sync` and the `syncrtld` events occurring during a `debug` and the ones occurring during a normal `run`.

Variables Valid for Specific Events

Certain variables are valid only for specific events as shown in the following tables.

TABLE 6-1 Variables Valid for `sig` Event

Variable	Description
`$sig`	Signal number that caused the event
`$sigstr`	Name of `$sig`
`$sigcode`	Subcode of `$sig` if applicable
`$sigcodestr`	Name of `$sigcode`
`$sigsender`	Process ID of sender of the signal, if appropriate

TABLE 6-2 Variable Valid for `exit` Event

Variable	Description
`$exitcode`	Value of the argument passed to `_exit(2)` or `exit(3)` or the return value of `main`

TABLE 6-3 Variable Valid for `dlopen` and `dlclose` Events

Variable	Description
`$dlobj`	Pathname of the load object dlopened or dlclosed

TABLE 6-4 Variables Valid for `sysin` and `sysout` Events

Variable	Description
`$syscode`	System call number
`$sysname`	System call name

TABLE 6-5 Variable Valid for `proc_gone` Events

`Variable`	`Description`
`$reason`	One of signal, exit, kill, or detach

Setting Event Handler Examples

The following are some examples of setting event handlers.

Setting a Watchpoint for Store to an Array Member

To set a watchpoint on `array[99]`, type:

```
(dbx) stop access w &array[99]
(2) stop access w &array[99], 4
(dbx) run
Running: watch.x2
watchpoint array[99] (0x2ca88[4]) at line 22 in file "watch.c"
   22array[i] = i;
```

Implementing a Simple Trace

To implement a simple trace, type:

```
(dbx) when step { echo at line $lineno; }
```

Enabling a Handler While Within a Function (`in` *function*)

To enable a handler while within a function, type:

```
<dbx> trace step -in foo
```

This is equivalent to:

```
# create handler in disabled state
when step -disable { echo Stepped to $line; }
t=$newhandlerid    # remember handler id
when in foo {
 # when entered foo enable the trace
 handler -enable "$t"
 # arrange so that upon returning from foo,
 # the trace is disabled.
 when returns { handler -disable "$t"; };
}
```

Determining the Number of Lines Executed

To see how many lines have been executed in a small program, type:

```
(dbx) stop step -count infinity # step and stop when count=inf
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

The program never stops—the program terminates. The number of lines executed is 133. This process is very slow. It is most useful with breakpoints on functions that are called many times.

Determining the Number of Instructions Executed by a Source Line

To count how many instructions a line of code executes, type:

```
(dbx) ...                        # get to the line in question
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx)status
(3) stop step -count 48/infinity # 48 instructions were executed
```

If the line you are stepping over makes a function call, the lines in the function are counted as well. You can use the `next` event instead of `step` to count instructions, excluding called functions.

Enabling a Breakpoint After An Event Occurs

Enable a breakpoint only after another event has occurred. For example, if your program begins to execute incorrectly in function `hash`, but only after the 1300'th symbol lookup, you would type:

```
(dbx) when in lookup -count 1300 {
    stop in hash
    hash_bpt=$newhandlerid
    when proc_gone -temp { delete $hash_bpt; }
}
```

Note – `$newhandlerid` is referring to the just executed `stop in` command.

Resetting Application Files for `replay`

If your application processes files that need to be reset during a `replay`, you can write a handler to do that each time you run the program:

```
(dbx) when sync { sh regen ./database; }
(dbx) run < ./database...# during which database gets clobbered
(dbx) save
...           # implies a RUN, which implies the SYNC event which
(dbx) restore   # causes regen to run
```

Checking Program Status

To see quickly where the program is while it is running, type:

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

Then type ^C to see a stack trace of the program without stopping it.

This is basically what the collector hand sample mode does (and more). Use SIGQUIT (^\) to interrupt the program because ^C is now used up.

Catch Floating Point Exceptions

To catch only specific floating point exceptions, for example, IEEE underflow, type:

```
(dbx) ignore FPE                # turn off default handler
(dbx) help signals | grep FPE  # can't remember the subcode name
...
(dbx) stop sig fpe FPE_FLTUND
...
```

CHAPTER 7

Using the Call Stack

This chapter discusses how `dbx` uses the *call stack,* and how to use the `where`, `hide`, `unhide`, and `pop` commands when working with the call stack.

The call stack represents all currently active routines—routines that have been called but have not yet returned to their respective caller.

Because the call stack grows from higher memory to lower memory, *up* means going toward the caller's frame (and eventually `main()`) and *down* means going toward the callee (and eventually the current function). The frame for the routine executing when the program stopped at a breakpoint, after a single-step, or when a fault occurs and produces a core file, is in lower memory. A caller routine, such as `main()`, is located in higher memory.

This chapter is organized into the following sections:

- Finding Your Place on the Stack
- Walking the Stack and Returning Home
- Moving Up and Down the Stack
- Popping the Call Stack
- Hiding Stack Frames

Finding Your Place on the Stack

Use the `where` command to find your current location on the stack.

```
where [-f] [-h] [-q] [-v] number_id
```

The `where` command is also useful for learning about the state of a program that has crashed and produced a core file. When this occurs, you can load the core file into `dbx`. (See "Debugging an Existing Core File" on page 12 and "Core File Debugging with dbx" in the Using dbx Commands section of the Sun WorkShop online help.)

For more information on the `where` command, see "where Command" in the Using dbx Commands section of the SunWorkShop online help.

Walking the Stack and Returning Home

Moving up or down the stack is referred to as "walking the stack." When you visit a function by moving up or down the stack, `dbx` displays the current function and the source line. The location from which you start , *home*, is the point where the program stopped executing. From home, you can move up or down the stack using the `up`, `down`, or `frame` commands.

The `dbx` commands `up` and `down` both accept a *number* argument that instructs `dbx` to move a number of frames up or down the stack from the current frame. If *number* is not specified, the default is 1. The `-h` option includes all hidden frames in the count.

Moving Up and Down the Stack

You can examine the local variables in functions other than the current one.

Moving Up the Stack

To move up the call stack (toward `main`) *number* levels:

```
up [-h] number
```

For more information, see "up Command" in the Using dbx Commands section of the Sun WorkShop online help.

Moving Down the Stack

To move down the call stack (toward the current stopping point) *number* levels:

```
down [-h] number
```

For more information, see "down Command" in the Using dbx Commands section of the Sun WorkShop online help.

You can also move around the call stack in the Sun WorkShop Debugging window. (See "Moving Around the Call Stack" in the Using the Debugging Window section of the Sun WorkShop online help.)

Moving to a Specific Frame

The `frame` command is similar to the `up` and `down` commands. It lets you go directly to the frame as given by numbers displayed by the `where` command.

```
frame
frame -h
frame [-h] number
frame [-h] +[number]
frame [-h] -[number]
```

The `frame` command without an argument displays the current frame number. With *number*, the command lets you go directly to the frame indicated by the number. By including a + (plus sign) or - (minus sign), the command lets you move an increment of one level up (+) or down (-). If you include a plus or minus sign with a *number*, you can move up or down the specified number of levels. The `-h` option includes any hidden frames in the count.

You can also move to a specific frame using the `pop` command (see "Popping the Call Stack" on page 94), or by using the Sun WorkShop Debugging window (see "Popping the Call Stack to the Current Frame" in the Using the Debugging Window section of the Sun WorkShop online help).

Popping the Call Stack

You can remove the stopped in function from the call stack, making the calling function the new stopped in function.

Unlike moving up or down the call stack, popping the stack changes the execution of your program. When the stopped in function is removed from the stack, it returns your program to its previous state, except for changes to global or static variables, external files, shared members, and similar global states.

The `pop` command removes one or more frames from the call stack. For example, to pop five frames from the stack, type:

```
pop 5
```

You can also pop to a specific frame. To pop to frame 5, type:

```
pop -f 5
```

For more information, see "pop Command" in the Using dbx Commands section of the Sun WorkShop online help.

You can also pop the call stack in the Sun WorkShop Debugging window. For more information, see "Popping the Call Stack One Function at a Time," "Popping the Call Stack to the Current Frame," and "Popping Debugger Frames from the Call Stack" in the Using dbx Commands section of the Sun WorkShop online help.

Hiding Stack Frames

Use the `hide` command to list the stack frame filters currently in effect.

To hide or delete all stack frames matching a regular expression, type:

```
hide [ regular_expression ]
```

The *regular_expression* matches either the function name, or the name of the loadobject, and is a `sh` or *ksh* file matching style regular expression.

Use `unhide` to delete all stack frame filters.

```
unhide 0
```

Because the `hide` command lists the filters with numbers, you can also use the `unhide` command with the filter number.

```
unhide [ number | regular_expression ]
```

CHAPTER 8

Evaluating and Displaying Data

In dbx, you can perform two types of data checking:

- Evaluate data (print) – Spot-checks the value of an expression
- Display data (display) – Monitors the value of an expression each time the program stops

This chapter is organized into the following sections:

- Evaluating Variables and Expressions
- Assigning a Value to a Variable
- Evaluating Arrays

Evaluating Variables and Expressions

This section discusses how to use dbx to evaluate variables and expressions.

Verifying Which Variable dbx Uses

If you are not sure which variable dbx is evaluating, use the which command to see the fully qualified name dbx is using.

To see other functions and files in which a variable name is defined, use the whereis command.

For information on the commands, see “which Command” and“where Command” in the Using dbx Commands section of the Sun WorkShop online help.

Variables Outside the Scope of the Current Function

When you want to evaluate or monitor a variable outside the scope of the current function:

- Qualify the name of the function. See "Qualifying Symbols With Scope Resolution Operators" on page 36.

or

- Visit the function by changing the current function. See "Visiting Code" on page 34.

Printing the Value of a Variable or an Expression

An expression should follow current language syntax, with the exception of the meta syntax that `dbx` introduces to deal with scope and arrays.

To evaluate a variable or expression, type:

```
print expression
```

For more information, see "print Command" in the Using dbx Commands section of the Sun WorkShop online help.

Note – `dbx` supports the C++ `dynamic_cast` and `typeid` operators. When evaluating expressions with these two operators, `dbx` makes calls to certain rtti functions made available by the compiler. If the source doesn't explicitly use the operators, those functions might not have been generated by the compiler, and `dbx` fails to evaluate the expression.

Printing C++

In C++ an object pointer has two types, its *static type* (what is defined in the source code) and its *dynamic type* (what an object was before any casts were made to it). `dbx` can sometimes provide you with the information about the dynamic type of an object.

In general, when an object has a virtual function table (a vtable) in it, `dbx` can use the information in the vtable to correctly determine an object's type.

You can use the `print` or `display` command with the `-r` (recursive) option. `dbx` displays all the data members directly defined by a class and those inherited from a base class.

These commands also take a `-d` or `+d` option that toggles the default behavior of the `dbx` environment variable `output_derived_type`.

Using the `-d` flag or setting the `dbx` environment variable `output_dynamic_type` to `on` when there is no process running generates a "program is not active" error message because it is not possible to access dynamic information when there is no process. An "illegal cast on class pointers" error message is generated if you try to find a dynamic type through a virtual inheritance. (Casting from a virtual base class to a derived class is not legal in C++.)

Evaluating Unnamed Arguments in C++ Programs

C++ lets you define functions with unnamed arguments. For example:

```
void tester(int)
{
};
main(int, char **)
{
   tester(1);
};
```

Though you cannot use unnamed arguments elsewhere in a program, the compiler encodes unnamed arguments in a form that lets you evaluate them. The form is as follows, where the compiler assigns an integer to `%n`:

```
_ARG%n
```

To obtain the name assigned by the compiler, type the `whatis` command with the function name as its target.

```
(dbx) whatis tester
void tester(int _ARG1);
(dbx) whatis main
int main(int _ARG1, char **_ARG2);
```

For more information, see “whatis Command” and “whatis Command Used With C++” in the Using dbx Commands section of the Sun WorkShop online help.

To evaluate (or display) an unnamed function argument, type:

```
(dbx) print _ARG1
_ARG1 = 4
```

Dereferencing Pointers

When you dereference a pointer, you ask for the contents of the container to which the pointer points.

To dereference a pointer, `dbx` displays the evaluation in the command pane; in this case, the value pointed to by `t`:

```
(dbx) print *t
*t = {
a = 4
}
```

You can also dereference a pointer in the Sun WorkShop Debugging window. See "Dereferencing a Pointer" in the Using the Debugging Window section of the Sun WorkShop online help.

Monitoring Expressions

Monitoring the value of an expression each time the program stops is an effective technique for learning how and when a particular expression or variable changes. The `display` command instructs `dbx` to monitor one or more specified expressions or variables. Monitoring continues until you turn it off with the `undisplay` command.

To display the value of a variable or expression each time the program stops, type:

```
display expression, ...
```

You can monitor more than one variable at a time. The `display` command used with no options prints a list of all expressions being displayed.

For more information, see "display Command" in the Using dbx Commands section of the Sun WorkShop online help.

You can also monitor the value of an expression in the Sun WorkShop Debugging window. See “Adding an Expression” in the Using the Debugging Window section of the Sun WorkShop online help.

Turning Off Display (Undisplay)

`dbx` continues to display the value of a variable you are monitoring until you turn off display with the `undisplay` command. You can turn off the display of a specified expression or turn off the display of all expressions currently being monitored.

To turn off the display of a particular variable or expression, type:

```
undisplay expression
```

To turn off the display of all currently monitored variables, type:

```
undisplay 0
```

For more information, see “undisplay Command” in the Using dbx Commands section of the Sun WorkShop online help.

You can turn off the display of a specified expression or all expressions in the Sun WorkShop Debugging window. See “Removing an Expression” in the Using the Debugging Window section of the Sun WorkShop online help.

Assigning a Value to a Variable

To assign a value to a variable, type:

```
assign variable = expression
```

You can also a value to a variable in the Sun WorkShop Debugging window. See “Changing Data Values” in the Using the Debugging Window section of the Sun WorkShop online help.

Evaluating Arrays

You evaluate arrays the same way you evaluate other types of variables.

Here is a sample Fortran array:

```
integer*4 arr(1:6, 4:7)
```

To evaluate the array, type:

```
print arr(2,4)
```

The `dbx print` command lets you evaluate part of a large array. Array evaluation includes:

- Array Slicing – Prints any rectangular, *n*-dimensional box of a multidimensional array.
- Array Striding – Prints certain elements only, in a fixed pattern, within the specified slice (which may be an entire array).

You can slice an array, with or without striding. (The default stride value is 1, which means print each element.)

Array Slicing

Array slicing is supported in the `print` and `display` commands for C, C++, and Fortran.

Array Slicing Syntax for C and C++

For each dimension of an array, the full syntax of the `print` command to slice the array is:

`print` *array-expression* [*first-expression* `..` *last-expression* `:` *stride-expression*]

where:

array-expression	Expression that should evaluate to an array or pointer type.
first-expression	First element to be printed. Defaults to 0.
last-expression	Last element to be printed. Defaults to upper bound.
stride-expression	Length of the stride (the number of elements skipped is *stride-expression*`-1`). Defaults to 1.

The first, last, and stride expressions are optional expressions that should evaluate to integers.

For example:

```
(dbx) print arr[2..4]
arr[2..4] =
[2] = 2
[3] = 3
[4] = 4
(dbx) print arr[..2]
arr[0..2] =
[0] = 0
[1] = 1
[2] = 2

(dbx) print arr[2..6:2]
arr[2..8:2] =
[2] = 2
[4] = 4
[6] = 6
```

Array Slicing Syntax for Fortran

For *each* dimension of an array, the full syntax of the `print` command to slice the array is:

```
print array-expression (first-expression .. last-expression : stride-expression)
```

where:

array-expression	Expression that should evaluate to an array pointer type.
first-expression	First element in a range, also first element to be printed. Defaults to lower bound.
last-expression	Last element in a range, but might not be the last element to be printed if stride is greater than 1. Defaults to upper bound.
stride-expression	Length of the stride (the number of elements skipped is *stride-expression*`-1`). Defaults to 1.

The first, last, and stride expressions are optional expressions that should evaluate to integers. For an *n*-dimensional slice, separate the definition of each slice with a comma.

For example:

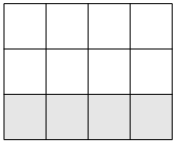
```
(dbx) print arr(2:6)
arr(2:6) =
(2) 2
(3) 3
(4) 4
(5) 5
(6) 6

(dbx) print arr(2:6:2)
arr(2:6:2) =
(2) 2
(4) 4
(6) 6
```

To specify rows and columns, type:

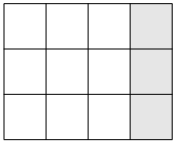
```
demo% f77 -g -silent ShoSli.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
    1   INTEGER*4 a(3,4), col, row
    2   DO row = 1,3
    3       DO col = 1,4
    4         a(row,col) = (row*10) + col
    5       END DO
    6   END DO
    7   DO row = 1, 3
    8        WRITE(*,'(4I3)') (a(row,col),col=1,4)
    9   END DO
    10  END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "ShoSli.f"
    7    DO row = 1, 3
```

To print row 3, type:



```
(dbx) print a(3:3,1:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (3,1)   31
        (3,2)   32
        (3,3)   33
        (3,4)   34
(dbx)
```

To print column 4, type:



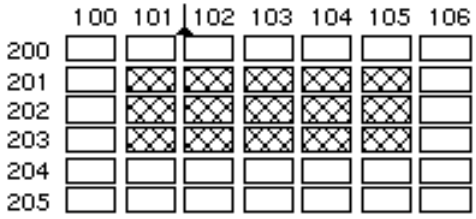
```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)
```

Slices

Here is an example of a two-dimensional, rectangular slice, with the default stride of 1 omitted.

```
print arr(201:203, 101:105)
```

This command prints a block of elements in a large array. Note that the command omits *stride-expression*, using the default stride value of 1.



The first two expressions (`201:203`) specify a slice in the first dimension of this two-dimensional array (the three-row column). The slice starts at the row `201` and ends with `203`. The second set of expressions, separated by a comma from the first, defines the slice for the 2nd dimension. The slice begins at column `101` and ends after column 105.

Strides

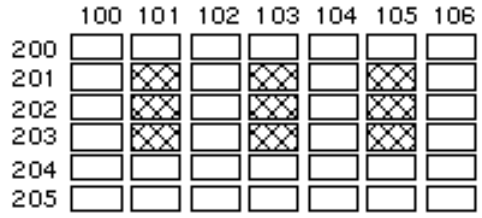
When you instruct `print` to *stride* across a slice of an array, `dbx` evaluates certain elements in the slice only, skipping over a fixed number of elements between each one it evaluates.

The third expression in the array slicing syntax, *stride-expression*, specifies the length of the stride. The value of *stride-expression* specifies the elements to print; the number of elements skipped is equal to *stride-expression*`-1`. The default stride value is 1, meaning: evaluate all of the elements in the specified slices.

Here is the same array used in the previous example of a slice. This time the print command includes a stride of 2 for the slice in the second dimension.

```
print arr(201:203, 101:105:2)
```

A stride of 2 prints every second element, skipping every other element.



For any expression you omit, print takes a default value equal to the declared size of the array. Here are examples showing how to use the shorthand syntax.

For a one-dimensional array, use the following commands:

`print arr`	Prints the entire array with default boundaries.
`print arr(:)`	Prints the entire array with default boundaries and default stride of 1.
`print arr(::`*stride-expression*`)`	Prints the entire array with a stride of *stride-expression.*

For a two-dimensional array, the following command prints the entire array.

```
print arr
```

To print every third element in the second dimension of a two-dimensional array, type:

```
print arr (:,::3)
```

CHAPTER 9

Using Runtime Checking

Runtime checking (RTC) lets you automatically detect runtime errors in an application during the development phase. Runtime checking lets you detect runtime errors such as memory access errors and memory leak errors and lets you monitor memory usage.

The following topics are covered in this chapter:

- Capabilities of Runtime Checking
- Using Runtime Checking
- Using Access Checking (SPARC only)
- Using Memory Leak Checking
- Using Memory Use Checking
- Suppressing Errors
- Using Runtime Checking on a Child Process
- Using Runtime Checking on an Attached Process
- Using Fix and Continue With Runtime Checking
- Runtime Checking Application Programming Interface
- Using Runtime Checking in Batch Mode
- Troubleshooting Tips

Note – Access checking is available only on SPARC systems.

Capabilities of Runtime Checking

Because runtime checking is an integral debugging feature, you can perform all debugging operations while using runtime checking except collecting performance data using the Sampling Collector.

Runtime checking:

- Detects memory access errors
- Detects memory leaks

- Collects data on memory use
- Works with all languages
- Works with multithreaded code
- Requires no recompiling, relinking, or makefile changes

Compiling with the `-g` flag provides source line number correlation in the runtime checking error messages. Runtime checking can also check programs compiled with the optimization `-O` flag. There are some special considerations with programs not compiled with the `-g` option.

You can use runtime checking by typing `dbx` command, or you can use the Sun WorkShop Debugging window and Runtime Checking window. See "Starting Runtime Checking" in the Using the Debugging Window section of the Sun WorkShop online help.

When to Use Runtime Checking

One way to avoid seeing a large number of errors at once is to use runtime checking earlier in the development cycle,—as you are developing the individual modules that make up your program. Write a unit test to drive each module and use runtime checking incrementally to check one module at a time. That way, you deal with a smaller number of errors at a time. When you integrate all of the modules into the full program, you are likely to encounter few new errors. When you reduce the number of errors to zero, you need to run runtime checking again only when you make changes to a module.

Runtime Checking Requirements

To use runtime checking, you must fulfill the following requirements:

- Programs compiled using a Sun compiler.
- Dynamic linking with `libc.`
- Use of the standard `libc malloc`, `free`, and `realloc` functions or allocators based on those functions. Runtime checking provides an application programming interface (API) to handle other allocators. See "Runtime Checking Application Programming Interface" on page 136.
- Programs that are not fully stripped; programs stripped with `strip -x` are acceptable.

Limitations

Runtime checking does not handle program text areas and data areas larger than 8 megabytes. For more information, see “Runtime Checking’s 8 Megabyte Limit” on page 138.

A possible solution is to insert special files in the executable image to handle program text areas and data areas larger than 8 megabytes.

Using Runtime Checking

To use runtime checking, enable the type of checking you want to use before you run the program.

Turning On Memory Use and Memory Leak Checking

To turn on memory use and memory leak checking, type:

```
(dbx) check -memuse
```

You can also turn on memory use and memory leak checking in the Sun WorkShop Debugging window. See “Displaying All Memory Leaks” in the Using the Debugging Window section of the Sun WorkShop online help.

When memory use checking or memory leak checking is turned on, the `showblock` command shows the details about the heap block at a given address. The details include the location of the block’s allocation and its size. For more information, see “showblock Command” in the Using dbx Commands section of the Sun WorkShop online help.

Turning On Memory Access Checking

To turn on memory access checking only, type:

```
(dbx) check -access
```

You can also turn on memory access checking in the Sun WorkShop Debugging window. See “Turning On Memory Access Checking” in the Using the Debugging Window section of the Sun WorkShop online help.

Turning On All Runtime Checking

To turn on memory leak, memory use, and memory access checking, type:

```
(dbx) check -all
```

For more information, see “check Command” in the Using dbx Commands section of the Sun WorkShop online help.

Turning Off Runtime Checking

To turn off runtime checking entirely, type:

```
(dbx) uncheck -all
```

For detailed information, see “uncheck Command” in the Using dbx Commands section of the Sun WorkShop online help.

Running Your Program

After turning on the types of runtime checking you want, run the program being tested, with or without breakpoints.

The program runs normally, but slowly because each memory access is checked for validity just before it occurs. If `dbx` detects invalid access, it displays the type and location of the error. Control returns to you (unless the `dbx` environment variable `rtc_auto_continue` is set to `on`. (See “rtc_auto_continue Environment Variable” in the Using dbx Commands section of the Sun WorkShop online help.))

You can then issue `dbx` commands, such as `where` to get the current stack trace or `print` to examine variables. If the error is not a fatal error, you can continue execution of the program with the `cont` command. The program continues to the next error or breakpoint, whichever is detected first. For detailed information, see “cont Command” in the Using dbx Commands section of the Sun WorkShop online help.

If `rtc_auto_continue` is set to `on`, runtime checking continues to find errors, and keeps running automatically. It redirects errors to the file named by the `dbx` environment variable `rtc_error_log_file_name.` (See "rtc_error_log_file_name Environment Variable" in the Using dbx Commands section of the Sun WorkShop online help.) The default log file name is `/tmp/dbx.errlog.`*uniqueid*.

You can limit the reporting of runtime checking errors using the `suppress` command. For detailed information, see "suppress Command" in the Using dbx Commands section of the Sun WorkShop online help.

Below is a simple example showing how to turn on memory access and memory use checking for a program called `hello.c`.

```
% cat -n hello.c
     1 #include <stdio.h>
     2 #include <stdlib.h>
     3 #include <string.h>
     4
     5 char *hello1, *hello2;
     6
     7 void
     8 memory_use()
     9 {
    10      hello1 = (char *)malloc(32);
    11      strcpy(hello1, "hello world");
    12      hello2 = (char *)malloc(strlen(hello1)+1);
    13      strcpy(hello2, hello1);
    14 }
    15
    16 void
    17 memory_leak()
    18 {
    19      char *local;
    20      local = (char *)malloc(32);
    21      strcpy(local, "hello world");
    22 }
    23
    24 void
    25 access_error()
    26 {
    27      int i,j;
    28
    29      i = j;
    30 }
    31
    32 int
    33 main()
    34 {
    35      memory_use();
    36      access_error();
    37      memory_leak();
    38      printf("%s\n", hello2);
    39      return 0;
    40 }
```

```
% cc -g -o hello hello.c
% dbx -C hello
Reading ld.so.1
Reading librtc.so
Reading libc.so.1
Reading libdl.so.1
(dbx) check -access
access checking - ON
(dbx) check -memuse
memuse checking - ON
(dbx) run
Running: hello
(process id 18306)
Enabling Error Checking... done
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff068
    which is 96 bytes above the current stack pointer
Variable is 'j'
Current function is access_error
   29       i = j;
(dbx) cont
hello world
Checking for memory leaks...
Actual leaks report   (actual leaks:       1 total size:     32 bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
                Address
======  ====== ==========  ======================================
    32       1    0x21aa8  memory_leak < main

Possible leaks report  (possible leaks:     0  total size:     0 bytes)

Checking for memory use...
Blocks in use report   (blocks in use:      2  total size:     44 bytes)

 Total  % of Num of  Avg     Allocation call stack
 Size    All Blocks  Size
======= ==== ====== ======  ======================================
     32  72%      1     32  memory_use < main
     12  27%      1     12  memory_use < main

execution completed, exit code is 0
```

The function `access_error()` reads variable `j` before it is initialized. Runtime checking reports this access error as a Read from uninitialized (`rui`).

The function `memory_leak()` does not free the variable `local` before it returns. When `memory_leak()` returns, this variable goes out of scope and the block allocated at line 20 becomes a leak.

The program uses global variables `hello1` and `hello2,` which are in scope all the time. They both point to dynamically allocated memory, which is reported as Blocks in use (`biu`).

Using Access Checking (SPARC only)

Access checking checks whether your program accesses memory correctly by monitoring each read, write, and memory free operation.

Programs might incorrectly read or write memory in a variety of ways; these are called memory access errors. For example, the program may reference a block of memory that has been deallocated through a `free()`call for a heap block. Or a function might return a pointer to a local variable, and when that pointer is accessed an error would result. Access errors might result in wild pointers in the program and can cause incorrect program behavior, including wrong outputs and segmentation violations. Some kinds of memory access errors can be very hard to track down.

Runtime checking maintains a table that tracks the state of each block of memory being used by the program. Runtime checking checks each memory operation against the state of the block of memory it involves and then determines whether the operation is valid. The possible memory states are:

- Unallocated, initial state – Memory has not been allocated. It is illegal to read, write, or free this memory because it is not owned by the program.
- Allocated, but uninitialized – Memory has been allocated to the program but not initialized. It is legal to write to or free this memory, but is illegal to read it because it is uninitialized. For example, upon entering a function, stack memory for local variables is allocated, but uninitialized.
- Read-only – It is legal to read, but not write or free, read-only memory.
- Allocated and initialized – It is legal to read, write, or free allocated and initialized memory.

Using runtime checking to find memory access errors is not unlike using a compiler to find syntax errors in your program. In both cases, a list of errors is produced, with each error message giving the cause of the error and the location in the program where the error occurred. In both cases, you should fix the errors in your program starting at the top of the error list and working your way down. One error can cause other errors in a chain reaction. The first error in the chain is, therefore, the “first cause,” and fixing that error might also fix some subsequent errors.

For example, a read from an uninitialized section of memory can create an incorrect pointer, which when dereferenced can cause another invalid read or write, which can in turn lead to yet another error.

Understanding the Memory Access Error Report

Runtime checking prints the following information for memory access errors:

Error	Information
`type`	Type of error.
`access`	Type of access attempted (read or write).
`size`	Size of attempted access.
`addr`	Address of attempted access.
`detail`	More detailed information about `addr`. For example, if `addr` is in the vicinity of the stack, then its position relative to the current stack pointer is given. If `addr` is in the heap, then the address, size, and relative position of the nearest heap block is given.
`stack`	Call stack at time of error (with batch mode).
`allocation`	If `addr` is in the heap, then the allocation trace of the nearest heap block are given.
`location`	Where the error occurred. If line number information is available, this information includes line number and function. If line numbers are not available, runtime checking provides function and address.

The following example shows a typical access error.

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xefffee50
    which is 96 bytes above the current stack pointer
Variable is ‘j’
Current function is rui
   12           i = j;
```

Memory Access Errors

Runtime checking detects the following memory access errors:

- `rui` (see “Read From Uninitialized Memory (`rui`) Error” on page 143)

- `rua` (see "Read From Unallocated Memory (`rua`) Error" on page 142)
- `wua` (see "Write to Unallocated Memory (`wua`) Error" on page 143)
- `wro` (see "Write to Read-Only Memory (`wro`) Error" on page 143)
- `mar` (see "Misaligned Read (`mar`) Error" on page 141)
- `maw` (see "Misaligned Write (`maw`) Error" on page 142)
- `duf` (see "Duplicate Free (`duf`) Error" on page 141)
- `baf` (see "Bad Free (`baf`) Error" on page 141)
- `maf` (see "Misaligned Free (`maf`) Error" on page 141)
- `oom` (see "Out of Memory (`oom`) Error" on page 142)

Note – Runtime checking does not perform array bounds checking and, therefore, does not report array bound violations as access errors.

Using Memory Leak Checking

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because there are no pointers pointing to the blocks, programs cannot reference them, much less free them. Runtime checking finds and reports such blocks.

Memory leaks result in increased virtual memory consumption and generally result in memory fragmentation. This might slow down the performance of your program and the whole system.

Typically, memory leaks occur because allocated memory is not freed and you lose a pointer to the allocated block. Here are some examples of memory leaks:

```
void
foo()
{
    char *s;
    s = (char *) malloc(32);

    strcpy(s, "hello world");

    return; /* no free of s. Once foo returns, there is no     */
           /* pointer pointing to the malloc'ed block,        */
            /* so that block is leaked.                       */
}
```

A leak can result from incorrect use of an API.

```
void
printcwd()
{

    printf("cwd = %s\n", getcwd(NULL, MAXPATHLEN));

    return; /* libc function getcwd() returns a pointer to    */
           /* malloc'ed area when the first argument is NULL, */
           /* program should remember to free this. In this   */
           /* case the block is not freed and results in leak.*/
}
```

You can avoid memory leaks by always freeing memory when it is no longer needed and paying close attention to library functions that return allocated memory. If you use such functions, remember to free up the memory appropriately.

Sometimes the term *memory leak* is used to refer to any block that has not been freed. This is a much less useful definition of a memory leak, because it is a common programming practice not to free memory if the program will terminate shortly. Runtime checking does not report a block as a leak, if the program still retains one or more pointers to it.

Detecting Memory Leak Errors

Runtime checking detects the following memory leak errors:

- `mel` (see "Memory Leak (`mel`) Error" on page 145)
- `air` (see "Address in Register (`air`) Error" on page 144)
- `aib` (see "Address in Block (`aib`) Error" on page 144)

Note – Runtime checking only finds leaks of `malloc` memory. If your program does not use `malloc`, runtime checking cannot find memory leaks.

Possible Leaks

There are two cases where runtime checking can report a "possible" leak. The first case is when no pointers are found pointing to the beginning of the block, but a pointer is found pointing to the *interior* of the block. This case is reported as an "Address in Block (`aib`)" error. If it was a stray pointer that pointed into the block, this would be a real memory leak. However, some programs deliberately move the

only pointer to an array back and forth as needed to access its entries. In this case, it would not be a memory leak. Because runtime checking cannot distinguish between these two cases, it reports both of them as possible leaks, letting you determine which are real memory leaks.

The second type of possible leak occurs when no pointers to a block are found in the data space, but a pointer is found in a register. This case is reported as an "Address in Register (`air`)" error. If the register points to the block accidentally, or if it is an old copy of a memory pointer that has since been lost, then this is a real leak. However, the compiler can optimize references and place the only pointer to a block in a register without ever writing the pointer to memory. Such a case would not be a real leak. Hence, if the program has been optimized and the report was the result of the `showleaks` command, it is likely not to be a real leak. In all other cases, it is likely to be a real leak. For more information, see "showleaks Command" in the Using dbx Commands section of the Sun WorkShop online help.

Note – Runtime leak checking requires the use of the standard `libc malloc/free/realloc` functions or allocators based on those functions. For other allocators, see "Runtime Checking Application Programming Interface" on page 136.

Checking for Leaks

If memory leak checking is turned on, a scan for memory leaks is automatically performed just before the program being tested exits. Any detected leaks are reported. The program should not be killed with the `kill` command. Here is a typical memory leak error message:

```
Memory leak (mel):
Found leaked block of size 6 at address 0x21718
At time of allocation, the call stack was:
    [1] foo() at line 63 in test.c
    [2] main() at line 47 in test.c
```

In the Sun WorkShop Runtime Checking window, clicking on the call stack location hypertext link takes you to that line of the source code in the editor window.

A UNIX program has a `main` procedure (called `MAIN` in f77) that is the top-level user function for the program. Normally, a program terminates either by calling `exit(3)` or by returning from `main`. In the latter case, all variables local to `main` go out of scope after the return, and any heap blocks they pointed to are reported as leaks (unless globals point to those same blocks).

It is a common programming practice not to free heap blocks allocated to local variables in `main`, because the program is about to terminate and return from `main` without calling `exit()`. To prevent runtime checking from reporting such blocks as memory leaks, stop the program just before `main` returns by setting a breakpoint on the last executable source line in `main`. When the program halts there, use the `showleaks` command to report all the true leaks, omitting the leaks that would result merely from variables in `main` going out of scope.

For more information, see "showleaks Command" in the Using dbx Commands section of the Sun WorkShop online help.

You can also check for memory leaks using the Sun WorkShop Debugging window. See "Displaying All Memory Leaks" and "Checking for New Memory Leaks" in the Using the Debugging Window section of the Sun WorkShop online help.

Understanding the Memory Leak Report

With leak checking turned on, you receive an automatic leak report when the program exits. All possible leaks are reported—provided the program has not been killed using the `kill` command. The level of detail in the report is controlled by the `dbx` environment variable `rtc_mel_at_exit` (see "rtc_mel_at_exit Environment Variable" in the Using dbx Commands section of the Sun WorkShop online help) or the Automatic leaks report at exit option in the Sun WorkShop Debugging Options dialog box (see "Generating an Automatic Leaks Report" in the Using the Debugging Window section of the Sun WorkShop online help). By default, a nonverbose leak report is generated.

Reports are sorted according to the combined size of the leaks. Actual memory leaks are reported first, followed by possible leaks. The verbose report contains detailed stack trace information, including line numbers and source files whenever they are available.

Both reports include the following information for memory leak errors:

location	Location where leaked block was allocated.
addr	Address of leaked block.
size	Size of leaked block.
stack	Call stack at time of allocation, as constrained by `check -frames`.

In the Sun WorkShop Runtime Checking window, the nonverbose report capsulizes the error information into a table, while the verbose report gives you a separate error message for each error. They both contain a hypertext link to the location of the error in the source code.

Here is the corresponding nonverbose memory leak report.

```
Actual leaks report   (actual leaks:   3 total size:   2427 bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
                Address
======  ====== ==========  ======================================
  1852       2      -      true_leak < true_leak
   575       1    0x22150  true_leak < main

Possible leaks report  (possible leaks:  1  total size:       8
bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
                Address
======  ====== ==========  ======================================
     8       1    0x219b0  in_block < main
```

Following is a typical verbose leak report.

```
Actual leaks report    (actual leaks:         3  total size:
2427 bytes)

Memory Leak (mel):
Found 2 leaked blocks with total size 1852 bytes
At time of each allocation, the call stack was:
    [1] true_leak() at line 220 in "leaks.c"
    [2] true_leak() at line 224 in "leaks.c"

Memory Leak (mel):
Found leaked block of size 575 bytes at address 0x22150
At time of allocation, the call stack was:
    [1] true_leak() at line 220 in "leaks.c"
    [2] main() at line 87 in "leaks.c"

Possible leaks report  (possible leaks:      1  total size:
8 bytes)

Possible memory leak -- address in block (aib):
Found leaked block of size 8 bytes at address 0x219b0
At time of allocation, the call stack was:
    [1] in_block() at line 177 in "leaks.c"
    [2] main() at line 100 in "leaks.c"
```

Generating a Leak Report

You can ask for a leak report at any time using the `showleaks` command, which reports new memory leaks since the last `showleaks` command. For more information, see "showleaks Command" in the using dbx Commands section of the Sun WorkShop online help.

Combining Leaks

Because the number of individual leaks can be very large, runtime checking automatically combines leaks allocated at the same place into a single combined leak report. The decision to combine leaks, or report them individually, is controlled by the `number-of-frames-to-match` parameter specified by the `-match` *m* option on a `check -leaks` or the `-m` option of the `showleaks` command. If the call stack at the time of allocation for two or more leaks matches to *m* frames to the exact program counter level, these leaks are reported in a single combined leak report.

Consider the following three call sequences:

Block 1	Block 2	Block 3
`[1] malloc`	`[1] malloc`	`[1] malloc`
`[2] d() at 0x20000`	`[2] d() at 0x20000`	`[2] d() at 0x20000`
`[3] c() at 0x30000`	`[3] c() at 0x30000`	`[3] c() at 0x31000`
`[4] b() at 0x40000`	`[4] b() at 0x41000`	`[4] b() at 0x40000`
`[5] a() at 0x50000`	`[5] a() at 0x50000`	`[5] a() at 0x50000`

If all of these blocks lead to memory leaks, the value of *m* determines whether the leaks are reported as separate leaks or as one repeated leak. If *m* is 2, Blocks 1 and 2 are reported as one repeated leak because the 2 stack frames above `malloc()` are common to both call sequences. Block 3 will be reported as a separate leak because the trace for `c()` does not match the other blocks. For *m* greater than 2, runtime checking reports all leaks as separate leaks. (The `malloc` is not shown on the leak report.)

In general, the smaller the value of *m*, the fewer individual leak reports and the more combined leak reports are generated. The greater the value of *m*, the fewer combined leak reports and the more individual leak reports are generated.

Fixing Memory Leaks

Once you have obtained a memory leak report, follow these guidelines for fixing the memory leaks.

- Most importantly, determine where the leak is. The leak report tells you the allocation trace of the leaked block, the place where the leaked block was allocated.
- You can then look at the execution flow of your program and see how the block was used. If it is obvious where the pointer was lost, the job is easy; otherwise you can use `showleaks` to narrow your leak window. By default the `showleaks` command gives you the new leaks created only since the last `showleaks` command. You can run `showleaks` repeatedly while stepping through your program to narrow the window where the block was leaked.

For more information, see "showleaks Command" in the using dbx Commands section of the Sun WorkShop online help.

Using Memory Use Checking

Memory use checking lets you see all the heap memory in use. You can use this information to get a sense of where memory is allocated in your program or which program sections are using the most dynamic memory. This information can also be useful in reducing the dynamic memory consumption of your program and might help in performance tuning

Memory use checking is useful during performance tuning or to control virtual memory use. When the program exits, a memory use report can be generated. Memory usage information can also be obtained at any time during program execution with the `showmemuse` command, which causes memory usage to be displayed. For information, see "showmemuse Command" in the Using dbx Commands section of the Sun WorkShop online help.

Turning on memory use checking also turns on leak checking. In addition to a leak report at the program exit, you also get a blocks in use (biu) report. By default, a nonverbose blocks in use report is generated at program exit. The level of detail in the memory use report is controlled by the `dbx` environment variable `rtc_biu_at_exit` (see "rtc_biu_at_exit Environment Variable" in the Using dbx Commands section of the Sun WorkShop online help) or the Automatic blocks report at exit option in the Sun WorkShop Debugging Options dialog box (see "Generating an Automatic Blocks Report" in the Using the Debugging Window section of the Sun WorkShop online help).

The following is a typical nonverbose memory use report.

```
Blocks in use report   (blocks in use: 5   total size:   40 bytes)

 Total  % of Num of  Avg     Allocation call stack
 Size    All Blocks  Size
======= ==== ====== ======  ====================================
     16  40%      2      8  nonleak < nonleak
      8  20%      1      8  nonleak < main
      8  20%      1      8  cyclic_leaks < main
      8  20%      1      8  cyclic_leaks < main
```

The following is the corresponding verbose memory use report:

```
Blocks in use report   (blocks in use: 5   total size:   40 bytes)

Block in use (biu):
Found 2 blocks totaling 16 bytes (40.00% of total; avg block size
8)
At time of each allocation, the call stack was:
     [1] nonleak() at line 182 in "memuse.c"
     [2] nonleak() at line 185 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21898 (20.00% of total)
At time of allocation, the call stack was:
     [1] nonleak() at line 182 in "memuse.c"
     [2] main() at line 74 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21958 (20.00% of total)
At time of allocation, the call stack was:
     [1] cyclic_leaks() at line 154 in "memuse.c"
     [2] main() at line 118 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21978 (20.00% of total)
At time of allocation, the call stack was:
     [1] cyclic_leaks() at line 155 in "memuse.c"
     [2] main() at line 118 in "memuse.c"
```

You can ask for a memory use report any time with the `showmemuse` command.

Suppressing Errors

Runtime checking provides a powerful error suppression facility that allows great flexibility in limiting the number and types of errors reported. If an error occurs that you have suppressed, then no report is given, and the program continues as if no error had occurred.

You can suppress errors using the `suppress` command (see "suppress Command" in the Using dbx Commands section of the Sun WorkShop online help). You can suppress the last reported error using the Suppress Last Reported Error button in the Sun WorkShop Runtime Checking window (see "Suppressing the Last Reported Error" in the Using the Debugging Window section of the Sun WorkShop online help).

You can undo error suppression using the `unsuppress` command (see "unsuppress Command" in the Using dbx Commands section of the Sun WorkShop online help).

Suppression is persistent across `run` commands within the same debug session, but not across `debug` commands.

Types of Suppression

The following types of suppression are available:

Suppression by Scope and Type

You must specify which type of error to suppress. You can specify which parts of the program to suppress. The options are:

Global	The default; applies to the whole program.
Load Object	Applies to an entire load object, such as a shared library, or the main program.
File	Applies to all functions in a particular file.
Function	Applies to a particular function.
Line	Applies to a particular source line.
Address	Applies to a particular instruction at an address.

Suppression of Last Error

By default, runtime checking suppresses the most recent error to prevent repeated reports of the same error. This is controlled by the `dbx` environment variable `rtc_auto_suppress`. When `rtc_auto_suppress` is set to `on` (the default), a particular access error at a particular location is reported only the first time it is encountered and suppressed thereafter. This is useful, for example, for preventing multiple copies of the same error report when an error occurs in a loop that is executed many times.

Limiting the Number of Errors Reported

You can use the `dbx` environment variable `rtc_error_limit` to limit the number of errors that will be reported. The error limit is used separately for access errors and leak errors. For example, if the error limit is set to 5, then a maximum of five access errors and five memory leaks are shown in both the leak report at the end of the run and for each `showleaks` command you issue. The default is 1000.

Suppressing Error Examples

In the following examples, `main.cc` is a file name, `foo` and `bar` are functions, and `a.out` is the name of an executable.

Do not report memory leaks whose allocation occurs in function `foo`.

```
suppress mel in foo
```

Suppress reporting blocks in use allocated from `libc.so.1.`

```
suppress biu in libc.so.1
```

Suppress read from uninitialized in all functions in `a.out`.

```
suppress rui in a.out
```

Do not report read from unallocated in file `main.cc`.

```
suppress rua in main.cc
```

Suppress duplicate free at line 10 of `main.cc`.

```
suppress duf at main.cc:10
```

Suppress reporting of all errors in function `bar`.

```
suppress all in bar
```

For more information, see “suppress Command” in the Using dbx Commands section of the Sun WorkShop online help.

Default Suppressions

To detect all errors, runtime checking does not require the program be compiled using the `-g` option (symbolic). However, symbolic information is sometimes needed to guarantee the correctness of certain errors, mostly `rui` errors. For this reason certain errors, `rui` for `a.out` and `rui`, `aib`, and `air` for shared libraries, are suppressed by default if no symbolic information is available. This behavior can be changed using the `-d` option of the `suppress` and `unsuppress` commands.

The following command causes runtime checking to no longer suppress read from uninitialized memory (`rui`) in code that does not have symbolic information (compiled without `-g`):

```
unsuppress -d rui
```

For more information, see “unsuppress Command” in the Using dbx Commands section of the Sun WorkShop online help

Using Suppression to Manage Errors

For the initial run on a large program, the large number of errors might be overwhelming. It might be better to take a phased approach. You can do so using the `suppress` command to reduce the reported errors to a manageable number, fixing just those errors, and repeating the cycle; suppressing fewer and fewer errors with each iteration.

For example, you could focus on a few error types at one time. The most common error types typically encountered are `rui`, `rua`, and `wua`, usually in that order. `rui` errors are less serious (although they can cause more serious errors to happen later).

Often a program might still work correctly with these errors. `rua` and `wua` errors are more serious because they are accesses to or from invalid memory addresses and always indicate a coding error.

You can start by suppressing `rui` and `rua` errors. After fixing all the `wua` errors that occur, run the program again, this time suppressing only `rui` errors. After fixing all the `rua` errors that occur, run the program again, this time with no errors suppressed. Fix all the `rui` errors. Lastly, run the program a final time to ensure no errors are left.

If you want to suppress the last reported error, use `suppress -last (`or in the Sun WorkShop Runtime Checking window, click the Suppress Last Reported Error button)`.`

Using Runtime Checking on a Child Process

To use runtime checking on a child process, you must have the `dbx` environment variable `rtc_inherit` set to `on`. By default, it is set to `off`. (See “rtc_inherit Environment Variable“ in the Using dbx Commands section of the Sun WorkShop online help.)

`dbx` supports runtime checking of a child process if runtime checking is enabled for the parent and the `dbx` environment variable `follow_fork_mode` is set to `child` (see “follow_fork_mode Environment Variable” in the Using dbx Commands section of the Sun WorkShop online help).

When a fork happens, `dbx` automatically performs runtime checking on the child. If the program calls `exec()`, the runtime checking settings of the program calling `exec()` are passed on to the program.

At any given time, only one process can be under runtime checking control. The following is an example.

```
% cat -n program1.c
     1 #include <sys/types.h>
     2 #include <unistd.h>
     3 #include <stdio.h>
     4
     5 int
     6 main()
     7 {
     8      pid_t child_pid;
     9      int parent_i, parent_j;
    10
    11      parent_i = parent_j;
    12
    13      child_pid = fork();
    14
    15      if (child_pid == -1) {
    16          printf("parent: Fork failed\n");
    17          return 1;
    18      } else if (child_pid == 0) {
    19          int child_i, child_j;
    20
    21          printf("child: In child\n");
    22          child_i = child_j;
    23          if (execl("./program2", NULL) == -1) {
    24              printf("child: exec of program2 failed\n");
    25              exit(1);
    26          }
    27      } else {
    28          printf("parent: child's pid = %d\n", child_pid);
    29      }
    30      return 0;
    31 }
%
```

```
% cat -n program2.c
     1
     2 #include <stdio.h>
     3
     4 main()
     5 {
     6      int program2_i, program2_j;
     7
     8      printf ("program2: pid = %d\n", getpid());
     9      program2_i = program2_j;
    10
    11      malloc(8);
    12
    13      return 0;
    14 }
%
```

RTC reports first error in the parent, program1

```
% cc -g -o program1 program1.c
% cc -g -o program2 program2.c
% dbx -C program1
Reading symbolic information for program1
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libc_psr.so.1
(dbx) check -all
access checking - ON
memuse checking - ON
(dbx) dbxenv follow_fork_mode child
(dbx) run
Running: program1
(process id 3885)
Enabling Error Checking... done
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff110
    which is 104 bytes above the current stack pointer
Variable is 'parent_j'
Current function is main
  11       parent_i = parent_j;
```

Because `follow_fork_mode` is set to `child`, when the fork occurs error checking is switched from the parent to the child process

RTC reports an error in the child

When the exec of `program2` occurs, the RTC settings are inherited by `program2` so access and memory use checking are enabled for that process

RTC reports an access error in the executed program, `program2`

```
(dbx) cont
dbx: warning: Fork occurred; error checking disabled in parent
detaching from process 3885
Attached to process 3886
stopped in _fork at 0xef6b6040
0xef6b6040: _fork+0x0008:bgeu    _fork+0x30
Current function is main
   13       child_pid = fork();
parent: child's pid = 3886
(dbx) cont
child: In child
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff108
    which is 96 bytes above the current stack pointer
Variable is 'child_j'
Current function is main
   22   child_i = child_j;
(dbx) cont
dbx: process 3886 about to exec("./program2")
dbx: program "./program2" just exec'ed
dbx: to go back to the original program use "debug $oprog"
Reading symbolic information for program2
Skipping ld.so.1, already read
Skipping librtc.so, already read
Skipping libc.so.1, already read
Skipping libdl.so.1, already read
Skipping libc_psr.so.1, already read
Enabling Error Checking... done
stopped in main at line 8 in file "program2.c"
    8       printf ("program2: pid = %d\n", getpid());
(dbx) cont
program2: pid = 3886
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff13c
    which is 100 bytes above the current stack pointer
Variable is 'program2_j'
Current function is main
    9       program2_i = program2_j;
(dbx) cont
Checking for memory leaks...
```

RTC prints a memory use and memory leak report for the process that exited while under RTC control, `program2`

```
Actual leaks report   (actual leaks:     1  total size:   8
bytes)

 Total  Num of  Leaked      Allocation call stack
 Size   Blocks  Block
               Address
======  ====== ========== ===================================
     8       1    0x20c50  main

Possible leaks report  (possible leaks:   0  total size:   0
bytes)

execution completed, exit code is 0
```

Using Runtime Checking on an Attached Process

Runtime checking works with attached processes. However, to use runtime checking on an attached process, the process must be started with `librtc.so` preloaded. `librtc.so` resides in the `lib` directory of Sun WorkShop, for example, if the product is installed in `/opt`, `librtc.so` is at `/opt/SUNWspro/lib`.

To preload `librtc.so`:

```
% setenv LD_PRELOAD path-to-librtc/librtc.so
```

Set `LD_PRELOAD` to preload `librtc.so` only when needed; do not keep it loaded all the time. For example:

```
% setenv LD_PRELOAD...
% start-your-application
% unsetenv LD_PRELOAD
```

Once you attach to the process, you can enable runtime checking.

If the program you want to attach to is forked or executed from some other program, you need to set `LD_PRELOAD` for the main program (which will fork). The setting of `LD_PRELOAD` is inherited across forks and execution.

Using Fix and Continue With Runtime Checking

You can use runtime checking along with fix and continue to isolate and fix programming errors rapidly. Fix and continue provides a powerful combination that can save you a lot of debugging time. Here is an example:.

```
% cat -n bug.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7      *s = 'c';
     8 }
     9
    10 main()
    11 {
    12      problem();
    13      return 0;
    14 }
% cat -n bug-fixed.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7
     8      s = (char *)malloc(1);
     9      *s = 'c';
    10 }
    11
    12 main()
    13 {
    14      problem();
    15      return 0;
    16 }
```

```
yourmachine46: cc -g bug.c
yourmachine47: dbx -C a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libintl.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libw.so.1
(dbx) check -access
access checking - ON
(dbx) run
Running: a.out
(process id 15052)
Enabling Error Checking... done
Write to unallocated (wua):
Attempting to write 1 byte through NULL pointer
Current function is problem
    7       *s = 'c';
(dbx) pop
stopped in main at line 12 in file "bug.c"
   12       problem();
(dbx) #at this time we would edit the file; in this example just
copy the correct version
(dbx) cp bug-fixed.c bug.c
(dbx) fix
fixing "bug.c" ......
pc moved to "bug.c":14
stopped in main at line 14 in file "bug.c"
   14       problem();
(dbx) cont

execution completed, exit code is 0
(dbx) quit
The following modules in `a.out' have been changed (fixed):
bug.c
Remember to remake program.
```

For more information on using fix and continue, see Chapter 11.

Runtime Checking Application Programming Interface

Both leak detection and access checking require that the standard heap management routines in the shared library `libc.so` be used so that runtime checking can keep track of all the allocations and deallocations in the program. Many applications write their own memory management routines either on top of the `malloc()` or `free()` `function` or stand-alone. When you use your own allocators (referred to as *private allocators*), runtime checking cannot automatically track them; thus you do not learn of leak and memory access errors resulting from their improper use.

However, runtime checking provides an API for the use of private allocators. This API allows the private allocators the same treatment as the standard heap allocators. The API itself is provided in the header file `rtc_api.h` and is distributed as a part of Sun WorkShop. The man page `rtc_api`(3x) details the runtime checking API entry points.

Some minor differences might exist with runtime checking access error reporting when private allocators do not use the program heap. The error report will not include the allocation item.

Using Runtime Checking in Batch Mode

The `bcheck` utility is a convenient batch interface to the runtime checking feature of `dbx`. It runs a program under `dbx` and by default, places the runtime checking error output in the default file `program.errs`.

The `bcheck` utility can perform memory leak checking, memory access checking, memory use checking, or all three. Its default action is to perform only leak checking. See the `bcheck`(1) man page for more details on its use.

`bcheck` Syntax

The syntax for `bcheck` is:

```
bcheck [-access | -all | -leaks | -memuse] [-o logfile] [-q]
[-s script] program [args]
```

Use the `-o` *logfile* option to specify a different name for the logfile. Use the `-s` *script* option before executing the program to read in the `dbx` commands contained in the file *script*. The *script* file typically contains commands like `suppress` and `dbxenv` to tailor the error output of the `bcheck` utility.

The `-q` option makes the `bcheck` utility completely quiet, returning with the same status as the program. This option is useful when you want to use the `bcheck` utility in scripts or makefiles.

`bcheck` Examples

To perform only leak checking on `hello`, type:

```
bcheck hello
```

To perform only access checking on `mach` with the argument `5`, type:

```
bcheck -access mach 5
```

To perform memory use checking on `cc` quietly and exit with normal exit status, type:

```
bcheck -memuse -q cc -c prog.c
```

The program does not stop when runtime errors are detected in batch mode. All error output is redirected to your error log file `logfile`. The program stops when breakpoints are encountered or if the program is interrupted.

In batch mode, the complete stack backtrace is generated and redirected to the error log file. The number of stack frames can be controlled using the `dbx` environment variable `stack_max_size`.

If the file `logfile` already exists, `bcheck` erases the contents of that file before it redirects the batch output to it.

Enabling Batch Mode Directly From dbx

You can also enable a batch-like mode directly from `dbx` by setting the `dbx` environment variables `rtc_auto_continue` and `rtc_error_log_file_name` (see "rtc_auto_continue Environment Variable" and "rtc_error_log_file_name Environment Variable" in the Using dbx Commands section of the Sun WorkShop online help).

If `rtc_auto_continue` is set to `on`, runtime checking continues to find errors and keeps running automatically. It redirects errors to the file named by the `dbx` environment variable `rtc_error_log_file_name.` (See "rtc_error_log_file_name Environment Variable" in the Using dbx Commands section of the Sun WorkShop online help.) The default log file name is `/tmp/dbx.errlog.`*uniqueid*. To redirect all errors to the terminal, set the `rtc_error_log_file_name` environment variable to `/dev/tty`.

By default, `rtc_auto_continue` is set to off.

Troubleshooting Tips

After error checking has been enabled for a program and the program is run, one of the following errors may be detected:

```
librtc.so and dbx version mismatch; Error checking disabled
```

This error can occur if you are using runtime checking on an attached process and have set `LD_PRELOAD` to a version of `librtc.so` other than the one shipped with your Sun WorkShop `dbx` image. To fix this, change the setting of `LD_PRELOAD`.

```
patch area too far (8mb limitation); Access checking disabled
```

Runtime checking was unable to find patch space close enough to a loadobject for access checking to be enabled. See "Runtime Checking's 8 Megabyte Limit" next.

Runtime Checking's 8 Megabyte Limit

When access checking is enabled, `dbx` replaces each load and store instruction with a branch instruction that branches to a patch area. This branch instruction has an 8 megabyte range. This means that if the debugged program has used up all the address space within 8 megabytes of the particular load or store instruction being replaced, no place exists to put the patch area.

If runtime checking cannot intercept all loads and stores to memory, it cannot provide accurate information and so disables access checking completely. Leak checking is unaffected.

`dbx` internally applies some strategies when it runs into this limitation and continues if it can rectify this problem. In some cases `dbx` cannot proceed; when this happens, it turns off access checking after printing an error message.

If you encounter this 8 megabyte limit, try the following workarounds.

1. Try using 32-bit SPARC V8 instead of 64-bit SPARC V9

   If you encounter the 8 megabyte problem with an application that is compiled with the `-xarch=v9` option, try doing your memory testing on a 32-bit version of the application. Because the 64-bit addresses require longer patch instruction sequences, using 32-bit addresses can alleviate the 8 megabyte problem. If this is not a good workaround, the following methods can be used on both 32-bit and 64-bit programs.

2. Try using traps.

   On UltraSparc hardware, `dbx` has a new ability to invoke a trap handler instead of using a branch. The transfer of control to a trap handler is significantly slower (up to 10 times), but it does not suffer from the 8 megabyte limit. The dbx `rtc_use_traps` environment variable is used to enable trap handlers (see "Setting `dbx` Environment Variables With the `dbxenv` Command" on page 27). It can be used on 32-bit and 64-bit programs, as long as the hardware is UltraSparc. You can check your hardware by using the system command `uname -m` and verifying that the result is `sun4u`.

3. Try adding patch area object files.

   You can use the `rtc_patch_area` shell script to create special `.o` files that can be linked into the middle of a large executable or shared library to provide more patch space. `See the rtc_patch_area`(1) man page.

   When `dbx` reaches the 8 megabyte limit, it tells you which load object was too large (the main program, or a shared library) and it prints out the total patch space needed for that load object.

   For the best results, the special patch object files should be evenly spaced throughout the executable or shared library, and the default size (8 megabytes) or smaller should be used. Also, do not add more than 10-20% more patch space than `dbx` says it requires. For example, if dbx says that it needs 31 megabytes for `a.out`, then add four object files created with the `rtc_patch_area` script, each one 8 megabytes in size, and space them approximately evenly throughout the executable.

When dbx finds explicit patch areas in an executable, it prints the address ranges spanned by the patch areas, which can help you to place them correctly on the link line.

4. Try dividing the large load object into smaller load objects.

   Split up the object files in your executable or your large library into smaller groups of object files. Then link them into smaller parts. If the large file is the executable, then split it up into a smaller executable and a series of shared libraries. If the large file is a shared library, then rearrange it into a set of smaller libraries.

   This technique allows dbx to find space for patch code in between the different shared objects.

5. Try adding a "pad" .so file.

   This should only be necessary if you are attaching to a process after it has started up.

   The runtime linker might place libraries so close together that patch space cannot be created in the gaps between the libraries. When dbx starts up the executable with runtime checking turned on, it asks the runtime linker to place an extra gap between the shared libraries, but when attaching to a process that was not started by dbx with runtime checking enabled, the libraries might be too close together.

   If this is occurs, (and if it is not possible to start the program using dbx) then you can try creating a shared library using the rtc_patch_area script and linking it into your program between the other shared libraries. See the rtc_patch_area(1) man page for more details.

Runtime Checking Errors

Errors reported by runtime checking generally fall in two categories. Access errors and leaks.

Access Errors

When access checking is turned on, runtime checking detects and reports the following types of errors.

Bad Free (`baf`) Error

Problem: Attempt to free memory that has never been allocated.

Possible causes: Passing a non-heap data pointer to `free()` or `realloc()`.

Example:

```
char a[4];
char *b = &a[0];

free(b);                        /* Bad free (baf) */
```

Duplicate Free (`duf`) Error

Problem: Attempt to free a heap block that has already been freed.

Possible causes: Calling `free()` more than once with the same pointer. In C++, using the delete operator more than once on the same pointer.

Example:

```
char *a = (char *)malloc(1);
free(a);
free(a);                        /* Duplicate free (duf) */
```

Misaligned Free (`maf`) Error

Problem: Attempt to free a misaligned heap block.

Possible causes: Passing an improperly aligned pointer to `free()` or `realloc()`; changing the pointer returned by `malloc`.

Example:

```
char *ptr = (char *)malloc(4);
ptr++;
free(ptr);                      /* Misaligned free */
```

Misaligned Read (`mar`) Error

Problem: Attempt to read data from an address without proper alignment.

Possible causes: Reading 2, 4, or 8 bytes from an address that is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:

```
char *s = “hello world”;
int *i = (int *)&s[1];
int j;

j = *i;                          /* Misaligned read (mar) */
```

Misaligned Write (maw) Error

Problem: Attempt to write data to an address without proper alignment.

Possible causes: Writing 2, 4, or 8 bytes to an address that is not half-word-aligned, word-aligned, or double-word-aligned, respectively.

Example:

```
char *s = “hello world”;
int *i = (int *)&s[1];

*i = 0;                          /* Misaligned write (maw) */
```

Out of Memory (oom) Error

Problem: Attempt to allocate memory beyond physical memory available.

Cause: Program cannot obtain more memory from the system. Useful in locating problems that occur when the return value from `malloc()` is not checked for `NULL`, which is a common programming mistake.

Example:

```
char *ptr = (char *)malloc(0x7fffffff);
/* Out of Memory (oom), ptr == NULL */
```

Read From Unallocated Memory (rua) Error

Problem: Attempt to read from nonexistent, unallocated, or unmapped memory.

Possible causes: A stray pointer, overflowing the bounds of a heap block or accessing a heap block that has already been freed.

Example:

```
char c, *a = (char *)malloc(1);
```

```
c = a[1];        /* Read from unallocated memory (rua) */
```

Read From Uninitialized Memory (`rui`) Error

Problem: Attempt to read from uninitialized memory.

Possible causes: Reading local or heap data that has not been initialized.

Example:

```
foo()
{   int i, j;
    j = i;       /* Read from uninitialized memory (rui) */
}
```

Write to Read-Only Memory (`wro`) Error

Problem: Attempt to write to read-only memory.

Possible causes: Writing to a text address, writing to a read-only data section (`.rodata`), or writing to a page that `mmap` has made read-only.

Example:

```
foo()
{   int *foop = (int *) foo;
    *foop = 0;              /* Write to read-only memory (wro) */
}
```

Write to Unallocated Memory (`wua`) Error

Problem: Attempt to write to nonexistent, unallocated, or unmapped memory.

Possible causes: A stray pointer, overflowing the bounds of a heap block, or accessing a heap block that has already been freed.

Example:

```
char *a = (char *)malloc(1);
a[1] = '\0';            /* Write to unallocated memory (wua) */
```

Memory Leak Errors

With leak checking turned on, runtime checking reports the following types of errors.

Address in Block (`aib`) Error

Problem: A possible memory leak. There is no reference to the start of an allocated block, but there is at least one reference to an address within the block.

Possible causes: The only pointer to the start of the block is incremented.

Example;

```
char *ptr;
main()
{
   ptr = (char *)malloc(4);
   ptr++;        /* Address in Block */
}
```

Address in Register (`air`) Error

Problem: A possible memory leak. An allocated block has not been freed, and no reference to the block exists anywhere in program memory, but a reference exists in a register.

Possible causes: This can occur legitimately if the compiler keeps a program variable only in a register instead of in memory. The compiler often does this for local variables and function parameters when optimization is turned on. If this error occurs when optimization has not been turned on, it is likely to be an actual memory leak. This can occur if the only pointer to an allocated block goes out of scope before the block is freed.

Example:

```
if (i == 0) {
       char *ptr = (char *)malloc(4);
       /* ptr is going out of scope */
}
  /* Memory Leak or Address in Register */
```

Memory Leak (`mel`) Error

Problem: An allocated block has not been freed, and no reference to the block exists anywhere in the program.

Possible causes: Program failed to free a block no longer used.

Example:

```
char *ptr;
    ptr = (char *)malloc(1);
    ptr = 0;
/* Memory leak (mel) */
```

CHAPTER 10

Data Visualization

If you need a way to display your data graphically as you debug your program from Sun WorkShop, you can use data visualization.

You can use data visualization during debugging to help you explore and comprehend large and complex data sets, simulate results, or interactively steer computations. The Data Graph window lets you "see" program data and analyze that data graphically. The graphs can be printed or printed to a file.

This chapter contains the following sections:

- Specifying Proper Array Expressions
- Automatic Updating of Array Displays
- Changing Your Display
- Analyzing Visualized Data
- Fortran Program Example
- C Program Example

Specifying Proper Array Expressions

To display your data, you must specify the array and how it should be displayed. You can open the Data Graph window from the Sun WorkShop Debugging window by typing an array name in the Expression text box. (See "Evaluating an Array" in the Using the Debugging Window section of the Sun WorkShop online help.) All scalar array types are supported except for complex (Fortran) array types.

Single-dimensional arrays are graphed (a *vector graph*) with the x-axis indicating the index and the y-axis indicating the array values. In the default graphic representation of a two-dimensional array (an *area graph*), the x-axis indicates the index of the first dimension, the y-axis indicates the index of the second dimension, while the z-axis represents the array values. You can visualize arrays of *n* dimensions, but at most, only two of those dimensions can vary.

You do not have to examine an entire data set. You can specify slices of an array, as shown in the following examples. FIGURE 10-1 and FIGURE 10-2 show the `bf` array from the sample Fortran program given at the end of this chapter.

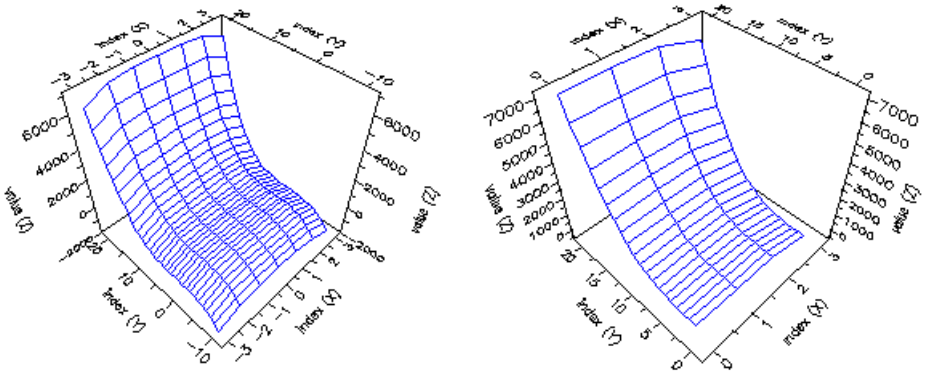


FIGURE 10-1 (left) `bf` Array Name Only; (right) `bf(0:3,1:20)`

The next two figures show the array with a range of values:

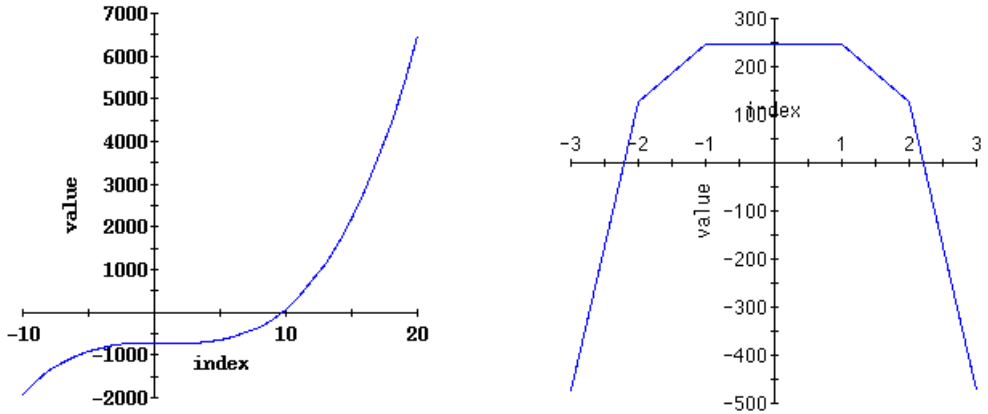


FIGURE 10-2 Array `bf`: (left) `bf(-3:3,:)`; (right) `bf(:,-7:7)`

Graphing an Array

You can compare different points during the execution of an application by graphing selected expressions in the source code. You can use the graph to observe where problems occur in the program. The Data Graph window enables you to graph arrays and examine different views of the same data. (See "Data Graph Window" in the Using the Debugging Window section of the Sun WorkShop online help.)

All arrays, except complex Fortran arrays, are supported.

Getting Ready

Before you can graph an array, you must:

1. **Load a program into the Sun WorkShop Debugging window by doing one of the following:**
   - Choose Debug ➤ New Program to load your program ("Debugging a Program New to Sun WorkShop" in the Using the Debugging Window section of the Sun WorkShop online help).
   - If the program was previously loaded in the current Sun WorkShop session, choose the program from the program list in the Debug menu (see "Debugging the Current Program" in the Using the Debugging WIndow section of the Sun WorkShop online help).
2. **Set at least one breakpoint in the program.**

   You can set a single breakpoint at the end of the program, or you can set one or more at points of interest in your program (see "Breaking at a Location" in the Using the Debugging Window section of the Sun WorkShop online help).
3. **Run your program.**

   (See "Starting Your Program Using Start" in the Using the Debugging Window section of the Sun WorkShop online help.)

   When the program stops at the breakpoint, decide which array you want to examine.

Multiple Ways to Graph an Array

Now you can graph the array. Sun WorkShop provides multiple ways to graph an array through Sun WorkShop:

- From the Debugging window, type an array namein the Expression text box and choose Data ➤ Graph Expression, or you can select an array in a text editor and choose Data ➤ Graph Selected in the Debugging window.

- From the Data Display tab or separate Data Display window, choose the Graph command from the Data menu or from the identical pop-up menu (right-click to open). If the array in the Data Display tab or window can be graphed, the Graph command is active.
- From the Data Graph window, choose Graph ➤ New, type an array name in the Expression text box in the Data Graph: New window, and click Graph.

  If you click the Replace current graph radio button and click Graph, the new graph replaces the current one. Otherwise, a new Data Graph window is opened.
- From the dbx Commands window, you can display a Data Graph directly from the dbx command line with the `vitem` command. (See "vitem Command" in the Using dbx Commands section of the Sun WorkShop online help.) Type:

```
(dbx) vitem -new array-expression
```

where:

array-expression specifies the array expression to be displayed.

Automatic Updating of Array Displays

The value of an array expression can change as the program runs. You can choose whether to show the array expression's new values at specific points within the program or at fixed time intervals in the Update section on the Data Grapher tab of the Sun WorkShop Debugging Options dialog box. (See "Data Grapher Defaults in the Using the Debugging Window section of the Sun WorkShop online help.)

If you want the values of an array updated each time the program stops at a breakpoint, you must turn on the Update at: Program stops option. As you step through the program, you can observe the change in array value at each stop. Use this option when you want to view the data change at each breakpoint. The default setting for this feature is off.

If you have a long-running program, choose the Update at: Fixed time interval option to observe changes in the array value over time. With a fixed time update, a timer is automatically set for every *n*th period. The default time is set for one second intervals. To change the default setting, choose Graph ➤ Default Options and change the time interval in the Debugging Options dialog box. (See "Displaying Updated Array Values" in the Using the Debugging Window section of the Sun WorkShop online help).

Note – Every time the timer reaches the nth period, Sun WorkShop tries to update the graph display; however, the array could be out of scope at that particular time and no update can be made.

Because the timer is also used in collecting data and when checking for memory leaks and access checking, you cannot use the Update at Fixed time interval setting when you are running the Sampling Collector or the runtime checking feature.

Changing Your Display

Once your data is graphically displayed, you can adjust and customize the display using the controls in the Data Graph window. This section presents examples of some of graph displays that you can examine.

The Data Graph window opens with the Magnify and Shrink options present when graphing any type of array. With an area graph, the window includes the Axis Rotation and Perspective Depth fields. These options let you change your view of the graph by rotating its axis or increasing or decreasing the depth perspective. To quickly rotate the graph, hold down the right mouse button with the cursor on the graph and drag to turn the graph. You can also enter the degree of rotation for each axis in the Axis Rotation field.

Click Show Options for more options. If the options are already shown, click Hide to hide the additional options.

FIGURE 10-3 shows two displays of the same array. The figure on the left shows the array with a Surface graph type with the Wire Mesh texture. The figure on the right shows the array with a Contour graph type delineated by range lines.

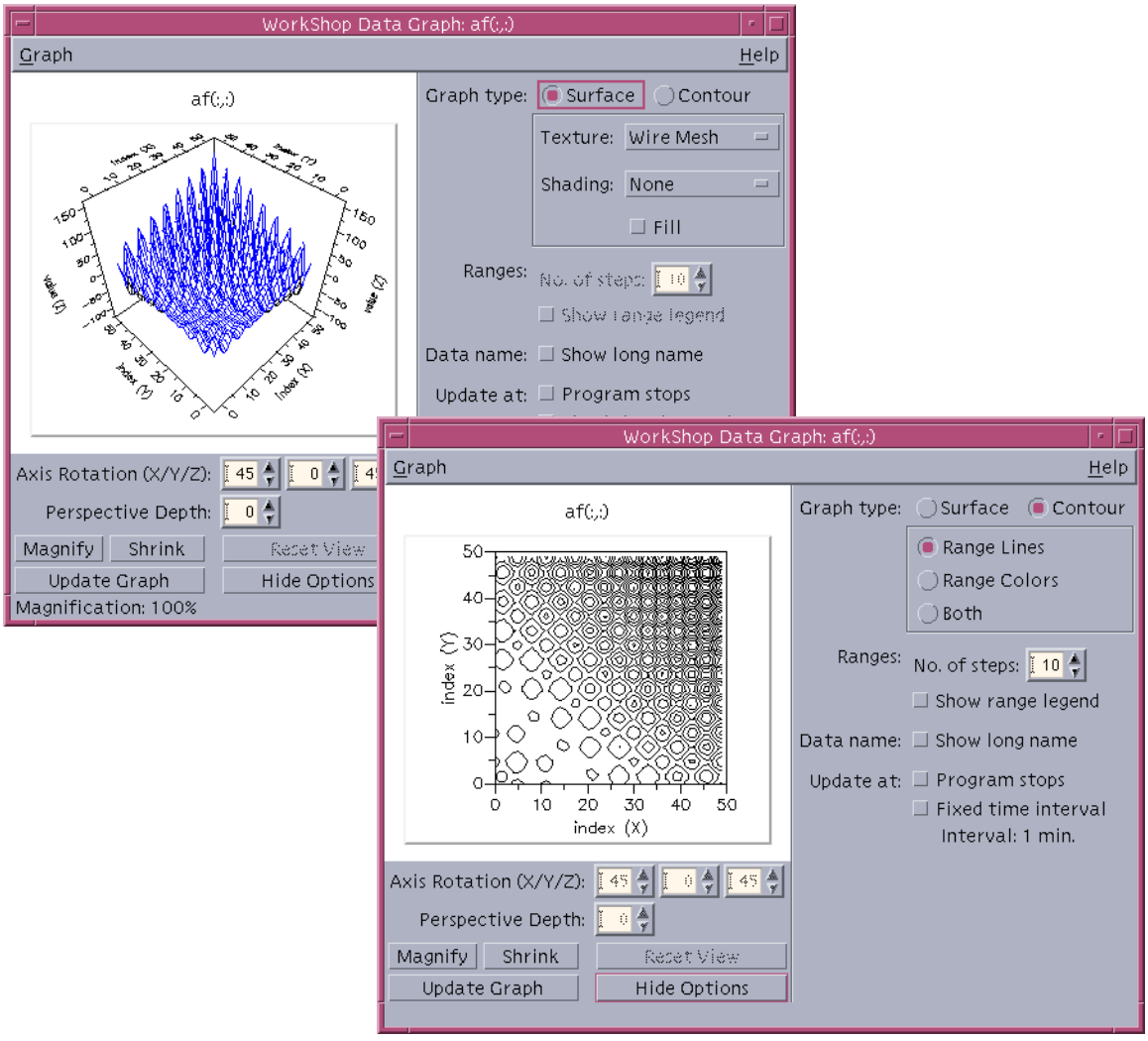


FIGURE 10-3 Two Displays of the Same Array

When you choose a Contour graph type, the range options let you see areas of change in the data values.

The display options for the Surface graph type are texture, shading, and fill.

Texture choices for the Surface graph type are Wire Mesh and Range Lines as shown in FIGURE 10-4.

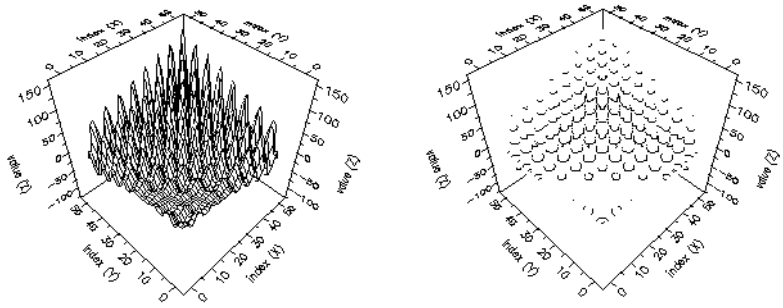


FIGURE 10-4 Surface graph with Wire Mesh (left) and Range Lines (right)

Shading choices for the Surface graph type are Light Source and Range Colors as shown in FIGURE 10-5.

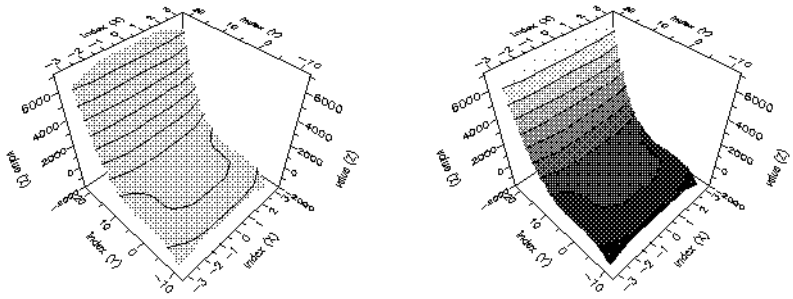


FIGURE 10-5 Surface Graph with Light Source (left) and Range Colors (right)

Turn on Fill to shade in areas of a graph or to create a solid surface graph as shown in FIGURE 10-6.

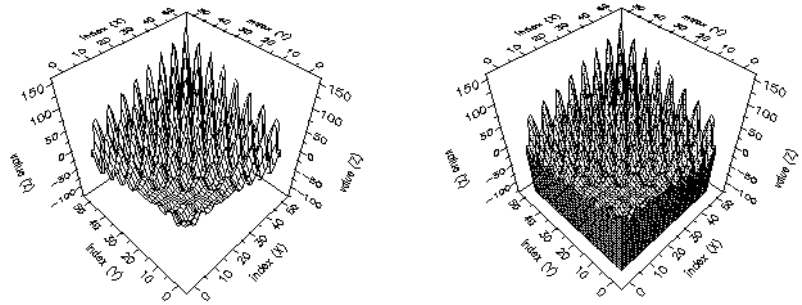


FIGURE 10-6 Surface Graph shaded in areas (left) or with a Solid Surface (right)

Choose Contour as the graph type to display an area graph using data range lines. With the Contour graph type, you have the additional options of displaying the graph in lines, in color, or both, as shown in FIGURE 10-7.

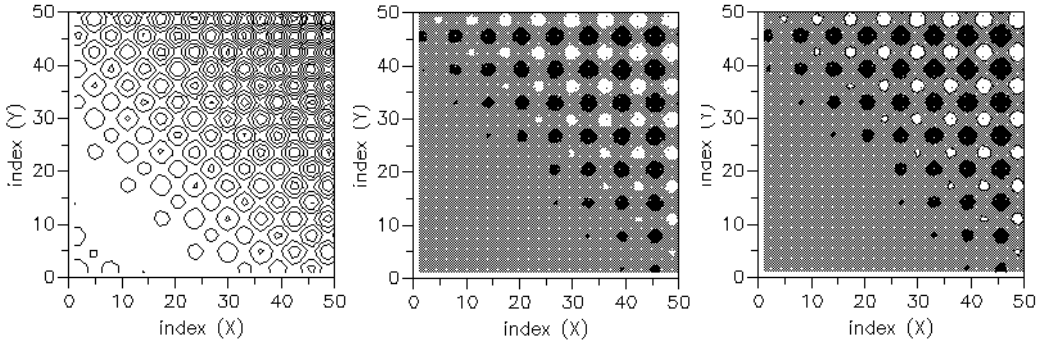


FIGURE 10-7 Contour Graph with Lines (left), in Color (center), or Both (right)

You can change the number of data value ranges being shown (see FIGURE 10-8) by changing the value in the Ranges: No. of steps text box.

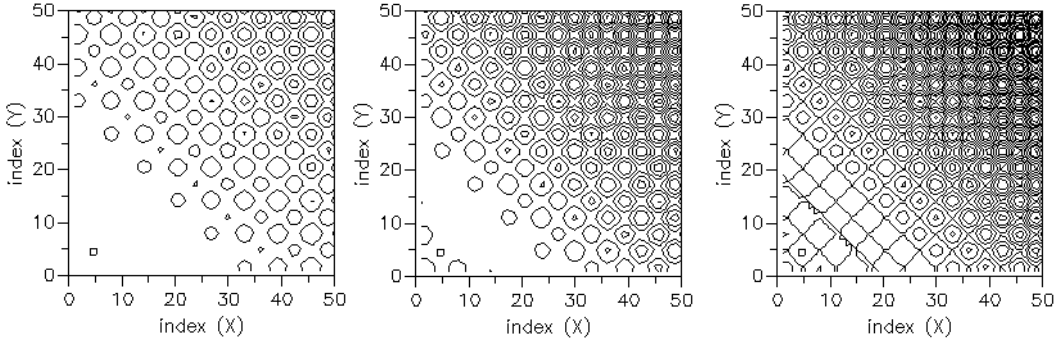


FIGURE 10-8 Contour Graph with Number of Steps (left to right): 5, 10, 15

If you choose a large number of steps, you can adversely affect the color map.

Click the Show range legend check box if you want a legend to display your range intervals.

Analyzing Visualized Data

There are different ways to update the data being visualized, depending on what you are trying to accomplish. For example, you can update on demand, at breakpoints, or at specified time intervals. You can observe changes or analyze final results. This section provides several scenarios illustrating different situations.

Two sample programs, `dg_fexamp` (Fortran) and `dg_cexamp` (C), are included with Sun WorkShop. These sample programs are used in the following scenarios to illustrate data visualization.

You can also use these programs for practice. They are located in *install-dir*`/Sunrises/WS6/examples/gatecrasher`, where the default *install-dir* is `/opt`. To use the programs, `change` to this directory and type `make`, the executable programs are created for you.

Scenario 1: Comparing Different Views of the Same Data

The same data can be graphed multiple times, letting you compare different graph types and different segments of data.

1. **Load the C or Fortran sample program.**
2. **Set a breakpoint at the end of the program.**

   (See "Breaking at a Location'" in the Using the Debugging Window section of the Sun WorkShop online help.)
3. **Start the program, running the program to that breakpoint.**

   (See "Starting Your Program Using Start" in the Using the Debugging Window section of the Sun WorkShop online help.)
4. **Type `bf` in the Expression text box in the Debugging window.**
5. **Choose Data ➤ Graph Expression.**

   A surface graph of both dimensions of the `bf` array is displayed.
6. **Choose Data ➤ Graph Expressions again to display a duplicate graph.**
7. **Click Show Options and click the Contour radio button for the graph type in one of the Data Graph windows.**

   Now you can compare different views of the same data (Surface versus Contour).

8. **Type `bf(1,:)` for the Fortran or `bf[1][..]` for the C example program in the Expression text box.**

9. **Choose Data ➤ Graph Expression to display a graph of a section of the data contained in the `bf` array.**

   You can now compare these different views of the data.

Scenario 2: Updating Graphs of Data Automatically

You can control the updating of a graph automatically by turning on the Update at: Program stops option on the Data Grapher tab of the Sun WorkShop Debugging Options dialog box. This feature lets you make comparisons of data as it changes during the execution of the program.

1. **Load the C or Fortran sample program.**

2. **Set a breakpoint at the end of the outer loop of the `bf` function.**

   (See "Breaking at a Location'" in the Using the Debugging Window section of the Sun WorkShop online help.)

3. **Start the program, running the program to that breakpoint.**

   (See "Starting Your Program Using Start" in the Using the Debugging Window section of the Sun WorkShop online help.)

4. **Type `bf` into the Expression text box in the Debugging window.**

5. **Choose Data ➤ Graph Expression.**

   A graph of the values in the `bf` array after the first loop iteration is displayed.

6. **Click Show Options and select the Update At: Program stops checkbox.**

7. **Choose Execute ➤ Continue to cause the execution of several other loop iterations of the program.**

   Each time the program stops at the breakpoint, the graph is updated with the values set in the previous loop iteration.

   Using the automatic update feature can save time when you want an up-to-date view of the data at each breakpoint.

Scenario 3: Comparing Data Graphs at Different Points in a Program

You can manually control the updating of a graph.

1. **Load the C or Fortran example program.**
2. **Set a breakpoint at the end of the outer loop of the `af` function.**

   (See "Breaking at a Location'" in the Using the Debugging Window section of the Sun WorkShop online help.)
3. **Start the program, running the program to that breakpoint.**

   (See "Starting Your Program Using Start" in the Using the Debugging Window section of the Sun WorkShop online help.)
4. **Type `af` in the Expression text box in the Debugging window.**
5. **Choose Data ➤ Graph Expression.**

   A graph of the values in the `af` array after the first loop iteration is displayed. Make sure automatic updating is turned off on this graph (the default setting).
6. **Execute another loop iteration of the program by choosing Execute ➤ Continue.**
7. **Display another graph of the `af` array by choosing Data ➤ Graph Expression.**

   This graph contains the data values set in the second iteration of the outer loop.

   You can now compare the data contained in the two loop iterations of the `af` array. You can use any graph with automatic updating turned off as a reference graph to a graph that is continually being updated automatically or manually.

Scenario 4: Comparing Data Graphs from Different Runs of the Same Program

Data graphs persist between different runs of the same program. Graphs from previous runs are not overwritten unless they are manually updated or automatic updating is turned on.

1. **Load the C or Fortran example program.**
2. **Set a breakpoint at the end of the program.**

   (See "Breaking at a Location'" in the Using the Debugging Window section of the Sun WorkShop online help.)

3. **Start the program, running the program to the breakpoint.**

   (See "Starting Your Program Using Start" in the Using the Debugging Window section of the Sun WorkShop online help.)

4. **Type `vec` in the Expression text box in the Debugging window.**

5. **Choose Data ➤ Graph Expression.**

   A graph of the `vec` array is displayed (as a sine curve).

6. **Now you can edit the program (for example, replace `sin` with `cos`).**

   Use fix and continue (see Chapter 11) to recompile the program and continue (click the Fix tool bar button).

7. **Restart the program.**

   Because automatic updating is turned off, the previous graph is not updated when the program reaches the breakpoint.

8. **Choose Data ➤ Graph Expression (`vec` is still in the Expression text box).**

   A graph of the current `vec` values is displayed beside the graph of the previous run.

   You can now compare the data from the two runs. The graph of the previous run changes only if it is updated manually using the update button or if automatic updating is turned on.

Fortran Program Example

```
real x,y,z,ct
real vec(100)
real af(50,50)
real bf(-3:3,-10:20)
real cf(50, 100, 200)

do x = 1,100
            ct = ct + 0.1
            vec(x) = sin(ct)
enddo

do x = 1,50
           do y = 1,50
              af(x,y) = (sin(x)+sin(y))*(20-abs(x+y))
           enddo
 enddo

do x = -3,3
          do y = -10,20
            bf(x,y) = y*(y-1)*(y-1.1)-10*x*x*(x*x-1)
          enddo
enddo

do x = 1,50
          do y = 1,100
            do z = 1,200
              cf(x,y,z) = 3*x*y*z - x*x*x - y*y*y - z*z*z
            enddo
          enddo
enddo

end
```

C Program Example

```
#include <math.h>
main()
{
    int x,y,z;
    float ct=0;
    float vec[100];
    float af[50][50];
    float bf[10][20];
    float cf[50][100][200];

    for (x=0; x<100; x++)
    {
          ct = ct + 0.1;
          vec[x] = sin(ct);
    }
    for (x=0; x<50; x++)
    {
          for (y=0; y<50; y++)
          {
          af[x][y] = (sin(x)+sin(y))*(20-abs(x+y));
          }
    }
    for (x=0; x<10; x++)
    {
          for (y=0; y<20; y++)
          {
            bf[x][y] = y*(y-1)*(y-1.1)-10*x*x*(x*x-1);
          }
    }
    for (x=0; x<50; x++)
    {
          for (y=0; y<100; y++)
          {
          for (z=0; z<200; z++)
          {
                  cf[x][y][z] = 3*x*y*z - x*x*x - y*y*y - z*z*z ;
          }
          }
    }
}
```

CHAPTER 11

Fixing and Continuing

Using the `fix` command lets you recompile edited source code quickly without stopping the debugging process.

This chapter is organized into the following sections:

- Using Fix and Continue
- Fixing Your Program
- Changing Variables After Fixing
- Modifying a Header File
- Fixing C++ Template Definitions

Using Fix and Continue

The fix and continue feature lets you modify and recompile a source file and continue executing without rebuilding the entire program. By updating the `.o` files and splicing them into your program, you don't need to relink.

The advantages of using fix and continue are:

- You do not have to relink the program.
- You do not have to reload the program for debugging.
- You can resume running the program from the fix location.

Note – Do not use the `fix` command if a build is in process; the output from the two processes will intermingle in the Sun WorkShop Building window.

How fix and continue Operates

Before using the `fix` command you need must edit the source in the editor window. (See “Modifying Source Using Fix and Continue” on page 162 for the ways you can modify your code). After saving changes, type `fix`. For information on the *fix* command, see “fix Command” in the Using dbx Commands section of the Sun WorkShop online help.

Once you have invoked the `fix` command, `dbx` calls the compiler with the appropriate compiler options. The modified files are compiled and shared object (`.so`) files are created. Semantic tests are done by comparing the old and new files.

The new object file is linked to your running process using the runtime linker. If the function on top of the stack is being fixed, the new stopped in function is the beginning of the same line in the new function. All the breakpoints in the old file are moved to the new file.

You can use fix and continue on files that have been compiled with or without debugging information, but there are some limitations in the functionality of the `fix` command and the `cont` command for files originally compiled without debugging information. See the `-g` option description in “fix Command” in the Using dbx Commands section of the Sun WorkShop online help for more information.

You can fix shared objects (`.so`) files, but they must be opened in a special mode. You can use either `RTLD_NOW|RTLD_GLOBAL` or `RTLD_LAZY|RTLD_GLOBAL` in the call to the `dlopen` function.

Modifying Source Using Fix and Continue

You can modify source code in the following ways when using fix and continue:

- Add, delete, or change lines of code in functions
- Add or delete functions
- Add or delete global and static variables

Problems can occur when functions are mapped from the old file to the new file. To minimize such problems when editing a source file:

- Do not change the name of a function.
- Do not add, delete, or change the type of arguments to a function.
- Do not add, delete, or change the type of local variables in functions currently active on the stack.
- Do not make changes to the declaration of a template or to template instances. Only the body of a C++ template function definition can be modified.

If you make any of the above changes, rebuild your entire program rather than using fix and continue.

Fixing Your Program

You can use the `fix` command to relink source files after you make changes, without recompiling the entire program. You can then continue execution of the program.

To fix your file:

1. **Save the changes to your source.**

   Sun WorkShop automatically saves your changes if you forget this step.

2. **Type `fix` at the `dbx` prompt.**

   Although you can do an unlimited number of fixes, if you have done several fixes in a row, consider rebuilding your program. The `fix` command changes the program image in memory, but not on the disk. As you do more fixes, the memory image gets out of sync with what is on the disk.

   The `fix` command does not make the changes within your executable file, but only changes the `.o` files and the memory image. Once you have finished debugging a program, you must rebuild your program to merge the changes into the executable. When you quit debugging, a message reminds you to rebuild your program.

   If you invoke the `fix` command with an option other than `-a` and without a file name argument, only the current modified source file is fixed.

   When `fix` is invoked, the current working directory of the file that was current at the time of compilation is searched before executing the compilation line. There might be problems locating the correct directory due to a change in the file system structure from compilation time to debugging time. To avoid this problem, use the command `pathmap`, which creates a mapping from one path name to another. Mapping is applied to source paths and object file paths.

Continuing After Fixing

You can continue executing using the `cont` command. (See “cont Command” in the Using dbx Commands section of the Sun WorkShop online help.)

Before resuming program execution, be aware of the following conditions that determine the effect of your changes.

Changing an Executed Function

If you made changes in a function that has already executed, the changes have no effect until:

- You run the program again
- That function is called the next time

If your modifications involve more than simple changes to variables, use the `fix` command, then the `run` command. Using the `run` command is faster because it does not relink the program.

Changing a Function Not Yet Called

If you have made changes in a function not yet called, the changes will be in effect when that function is called.

Changing a Function Currently Being Executed

If you have made changes to the function currently being executed, the impact of the `fix` command depends on where the change is relative to the stopped in function:

- If the change is in code that has already been executed, the code is not re-executed. Execute the code by popping the current function off the stack (see "pop Command" in the Using dbx Commands section of the Sun WorkShop online help and "Popping the Call Stack to the Current Frame" in the Using the Debugging Window section of the Sun WorkShop online help) and continuing from where the changed function is called. You need to know your code well enough to determine whether the function has side effects that can't be undone (for example, opening a file).
- If the change is in code that is yet to be executed, the new code is run.

Changing a Function Presently on the Stack

If you have made changes to a function presently on the stack, but not to the stopped in function, the changed code is not used for the present call of that function. When the stopped in function returns, the old versions of the function on the stack are executed.

There are several ways to solve this problem:

- Use the `pop` command to pop the stack until all changed functions are removed from the stack. You need to know your code to be sure that no problems are created.
- Use the `cont at` *linenum* command to continue from another line.

- Manually repair data structures (use the `assign` command) before continuing.
- Rerun the program using the `run` command.

If there are breakpoints in modified functions on the stack, the breakpoints are moved to the new versions of the functions. If the old versions are executed, the program does not stop in those functions.

Changing Variables After Fixing

Changes made to global variables are not undone by the `pop` command or the `fix` command. To reassign correct values to global variables manually, use the `assign` command. (See “assign Command” in the Using dbx Commands section of the Sun WorkShop online help.)

The following example shows how a simple bug can be fixed. The application gets a segmentation violation in line 6 when trying to dereference a NULL pointer.

```
dbx[1] list 1,$
    1#include <stdio.h>
    2
    3char *from = “ships”;
    4void copy(char *to)
    5{
    6   while ((*to++ = *from++) != ’\0’);
    7   *to = ’\0’;
    8}
    9
    10main()
    11{
    12  char buf[100];
    13
    14  copy(0);
    15  printf("%s\n", buf);
    16  return 0;
    17}
(dbx) run
Running: testfix
(process id 4842)
signal SEGV (no mapping at the fault address) in copy at line 6
in file “testfix.cc”
    6   while ((*to++ = *from++) != ’\0’);
```

Change line 14 to `copy` to `buf` instead of `0` and save the file, then do a fix:

```
    14  copy(buf);  <=== modified line
(dbx) fix
fixing “testfix.cc” .....
pc moved to “testfix.cc”:6
stopped in copy at line 6 in file “testfix.cc”
    6   while ((*to++ = *from++) != ’\0’)
```

If the program is continued from here, it still gets a segmentation fault because the zero-pointer is still pushed on the stack. Use the `pop` command to pop one frame of the stack:

```
(dbx) pop
stopped in main at line 14 in file “testfix.cc”
      14 copy(buf);
```

If the program is continued from here, it runs, but does not print the correct value because the global variable `from` has already been incremented by one. The program would print `hip`s and not `ships`. Use the `assign` command to restore the global variable and then use the `cont command`. Now the program prints the correct string:

```
(dbx) assign from = from-1
(dbx) cont
ships
```

Modifying a Header File

Sometimes it may be necessary to modify a header (`.h`) file as well as a source file. To be sure that the modified header file is accessed by all source files in the program that include it, you must give as an argument to the `fix` command a list of all the source files that include that header file. If you do not include the list of source files, only the primary source file is recompiled and only it includes the modified version of the header file. Other source files in the program continue to include the original version of that header file.

Fixing C++ Template Definitions

C++ template definitions cannot be fixed directly. Fix the files with the template instances instead. You can use the `-f` option to overwrite the date-checking if the template definition file has not changed. `dbx` looks for template definition `.o` files in the default repository directory `SunWS_cache`. The `-ptr` compiler switch is not supported by the `fix` command in `dbx`.

CHAPTER 12

Debugging Multithreaded Applications

`dbx` can debug multithreaded applications that use either Solaris threads or POSIX threads. With `dbx`, you can examine stack traces of each thread, resume all threads, `step` or `next` a specific thread, and navigate between threads.

`dbx` recognizes a multithreaded program by detecting whether it utilizes `libthread.so`. The program will use `libthread.so` either by explicitly being compiled with `-lthread` or `-mt`, or implicitly by being compiled with `-lpthread`.

This chapter describes how to find information about and debug threads using the `dbx thread` commands.

This chapter is organized into the following sections:

- Understanding Multithreaded Debugging
- Understanding LWP Information

Understanding Multithreaded Debugging

When it detects a multithreaded program, `dbx` tries to load `libthread_db.so`, a special system library for thread debugging located in `/usr/lib`.

`dbx` is synchronous; when any thread or lightweight process (LWP) stops, all other threads and LWPs sympathetically stop. This behavior is sometimes referred to as the "stop the world" model.

Note – For information on multithreaded programming and LWPs, see the Solaris *Multithreaded Programming Guide*.

Thread Information

The following thread information is available in `dbx`:

```
(dbx) threads
    t@1 a l@1  ?()  running   in main()
    t@2      ?() asleep on 0xef751450  in_swtch()
    t@3 b l@2  ?()  running in sigwait()
    t@4     consumer()  asleep on 0x22bb0 in _lwp_sema_wait()
  *>t@5 b l@4 consumer()  breakpoint     in Queue_dequeue()
    t@6 b l@5 producer()     running       in _thread_start()
(dbx)
```

Each line of information is composed of the following:

- The * (asterisk) indicates that an event requiring user attention has occurred in this thread. Usually this is a breakpoint.

  An 'o' instead of an asterisk indicates that a `dbx` internal event has occurred.
- The > (arrow) denotes the current thread.
- *t@num*, the thread id, refers to a particular thread. The *number* is the `thread_t` value passed back by `thr_create`.
- b l@*num* or a l@*num* means the thread is bound to or active on the designated LWP, meaning the thread is actually runnable by the operating system.
- The "Start function" of the thread as passed to `thr_create`. A `?()` means that the start function is not known.
- The thread state (See TABLE 12-1 for descriptions of the thread states.)
- The function that the thread is currently executing.

TABLE 12-1 Thread and LWP States

Thread and LWP States	Description
suspended	The thread has been explicitly suspended.
runnable	The thread is runnable and is waiting for an LWP as a computational resource.
zombie	When a detached thread exits (`thr_exit()`), it is in a zombie state until it has rejoined through the use of `thr_join()`. `THR_DETACHED` is a flag specified at thread creation time (`thr_create()`). A non-detached thread that exits is in a zombie state until it has been reaped.

TABLE 12-1 Thread and LWP States *(Continued)*

Thread and LWP States	Description
asleep on *syncobj*	Thread is blocked on the given synchronization object. Depending on what level of support `libthread` and `libthread_db` provide, *syncobj* might be as simple as a hexadecimal address or something with more information content.
active	The thread is active on an LWP, but `dbx` cannot access the LWP.
unknown	`dbx` cannot determine the state.
lwpstate	A bound or active thread state has the state of the LWP associated with it.
running	LWP was running but was stopped in synchrony with some other LWP.
syscall *num*	LWP stopped on an entry into the given system call #.
syscall return *num*	LWP stopped on an exit from the given system call #.
job control	LWP stopped due to job control.
LWP suspended	LWP is blocked in the kernel.
single stepped	LWP has just completed a single step.
breakpoint	LWP has just hit a breakpoint.
fault *num*	LWP has incurred the given fault #.
signal *name*	LWP has incurred the given signal.
process sync	The process to which this LWP belongs has just started executing.
LWP death	LWP is in the process of exiting.

Viewing the Context of Another Thread

To switch the viewing context to another thread, use the `thread` command. The syntax is:

```
thread [-info] [-hide] [-unhide] [-suspend] [-resume] tid
```

To display the current thread, type:

```
thread
```

To switch to thread *tid*, type:

```
thread tid
```

For more information on the `thread` command, see “thread Command” in the Using dbx Commands section of the Sun WorkShop online help.

Viewing the Threads List

The following are commands for viewing the `threads` list. The syntax is:

```
threads [-all} [-mode [all|filter] [auto|manual]]
```

To print the list of all known threads, type:

```
threads
```

To print threads normally not printed (zombies), type:

```
threads -all
```

For more information on the `threads` command, see “threads Command” in the Using dbx Commands section of the Sun WorkShop online help.

Resuming Execution

Use the `cont` command to resume program execution. Currently, threads use synchronous breakpoints, so all threads resume execution.

Understanding LWP Information

Normally, you need not be aware of LWPs. There are times, however, when thread level queries cannot be completed. In these cases, use the `lwps` command to show information about LWPs.

```
(dbx) lwps
    l@1 running in main()
    l@2 running in sigwait()
    l@3 running in _lwp_sema_wait()
  *>l@4 breakpoint in Queue_dequeue()
    l@5 running in _thread_start()
(dbx)
```

Each line of the LWP list contains the following:

- The * (asterisk) indicates that an event requiring user attention has occurred in this LWP.
- The arrow denotes the current LWP.
- l@*num* refers to a particular LWP.
- The next item represents the LWP state.
- in *func_name()* identifies the function that the LWP is currently executing.

CHAPTER 13

Debugging Child Processes

This chapter describes how to debug a child process. `dbx` has several facilities to help you debug processes that create children using the `fork` (2) and `exec` (2) functions.

This chapter is organized into the following sections:

- Attaching to Child Processes
- Following the `exec` Function
- Following the `fork` Function
- Interacting with Events

Attaching to Child Processes

You can attach to a running child process in one of the following ways.

When starting `dbx`:

```
$ dbx progname pid
```

From the command line:

```
(dbx) debug progname pid
```

You can substitute *progname* with the name - (minus sign), so that `dbx` finds the executable associated with the given process id (*pid*). After using a `-`, a subsequent `run` command or `rerun` command does not work because `dbx` does not know the full path name of the executable.

You can also attach to a running child process using the Sun WorkShop Debugging window. (See "Attaching to a Running Process" in the Using the Debugging window section of the Sun WorkShop online help.)

Following the exec Function

If a child process executes a new program using the exec(2)function or one of its variations, the process id does not change, but the process image does. dbx automatically takes note of a call to the exec() function and does an implicit reload of the newly executed program.

The original name of the executable is saved in $oprog. To return to it, use debug $oprog.

Following the fork Function

If a child process calls the vfork(), fork(1), or fork(2) function, the process id changes, but the process image stays the same. Depending on how the dbx environment variable follow_fork_mode is set, dbx does one of the following.

Parent	In the traditional behavior, dbx ignores the fork and follows the parent.
Child	dbx automatically switches to the forked child using the new process ID. All connection to and awareness of the original parent is lost.
Both	This mode is available only when using dbx through Sun WorkShop.
Ask	You are prompted to choose parent, child, both, or stop to investigate whenever dbx detects a fork. If you choose stop, you can examine the state of the program, then type cont to continue; you will be prompted to select which way to proceed.

Interacting with Events

All breakpoints and other events are deleted for any exec() or fork() process. You can override the deletion for forked processes by setting the dbx environment variable follow_fork_inherit to on, or make the events permanent using the -perm eventspec modifier. For more information on using event specification modifiers, see Chapter 6.

CHAPTER 14

Working With Signals

This chapter describes how to use `dbx` to work with signals. `dbx` supports the `catch` command, which instructs `dbx` to stop a program when `dbx` detects any of the signals appearing on the catch list.

The `dbx` commands `cont`, `step`, and `next` support the `-sig` *signal_name* option, which lets you resume execution of a program with the program behaving as if it had received the signal specified in the `cont -sig` command.

This chapter is organized into the following sections.

- Understanding Signal Events
- Catching Signals
- Sending a Signal in a Program
- Automatically Handling Signals

Understanding Signal Events

When a signal is to be delivered to a process that is being debugged, the signal is redirected to `dbx` by the kernel. When this happens, you usually receive a prompt. You then have two choices:

- "Cancel" the signal when the program is resumed—the default behavior of the `cont` command—facilitating easy interruption and resumption with `SIGINT` (Control-C) as shown in FIGURE 14-1.

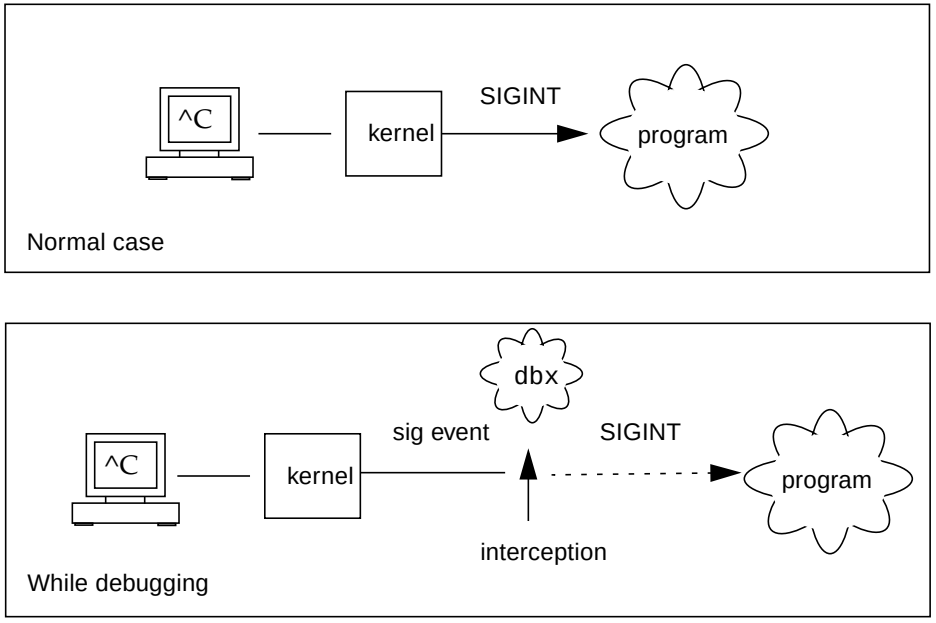


FIGURE 14-1 Intercepting and Cancelling the SIGINT Signal

- "Forward" the signal to the process using:

```
cont -sig signal
```

In addition, if a certain signal is received frequently, you can arrange for `dbx` to forward automatically the signal because you do not want it displayed:

```
ignore signal # "ignore"
```

However, the signal is still forwarded to the process. A default set of signals is automatically forwarded in this manner (see "ignore Command" in the Using dbx Commands section of the Sun WorkShop online help).

Catching Signals

By default, the catch list contains many of the more than 33 detectable signals. (The numbers depend upon the operating system and version.) You can change the default catch list by adding signals to or removing them from the default catch list.

To see the list of signals currently being trapped, type `catch` with no *signal* argument.

```
(dbx) catch
```

To see a list of the signals currently being *ignored* by `dbx` when the program detects them, type `ignore` with no *signal* argument.

```
(dbx) ignore
```

Changing the Default Signal Lists

You control which signals cause the program to stop by moving the signal names from one list to the other. To move signal names, supply a signal name that currently appears on one list as an argument to the other list.

For example, to move the `QUIT` and `ABRT` signals from the catch list to the ignore list:

```
(dbx) ignore QUIT ABRT
```

Trapping the FPE Signal

Often programmers working with code that requires floating point calculations want to debug exceptions generated in a program. When a floating point exception like overflow or divide by zero occurs, the system returns a reasonable answer as the result for the operation that caused the exception. Returning a reasonable answer lets the program continue executing quietly. Solaris implements the IEEE Standard for Binary Floating Point Arithmetic definitions of reasonable answers for exceptions.

Because a reasonable answer for floating point exceptions is returned, exceptions do not automatically trigger the signal SIGFPE.

To find the cause of an exception, you need to set up a trap handler in the program so that the exception triggers the signal SIGFPE. (See ieee_handler(3m) man page for an example of a trap handler.)

You can enable a trap using:

- ieee_handler
- fpsetmask (see the fpsetmask(3c) man page)
- -ftrap compiler flag (for FORTRAN 77 and Fortran 95, see the f77(1) and f95(1) man pages)

When you set up a trap handler using the ieee_handler command, the trap enable mask in the hardware floating point status register is set. This trap enable mask causes the exception to raise the SIGFPE signal at run time.

Once you have compiled the program with the trap handler, load the program into dbx. Before you can catch the SIGFPE signal, you must add FPE to the dbx signal catch list.

```
(dbx) catch FPE
```

By default, FPE is on the ignore list.

Determining Where the Exception Occurred

After adding FPE to the catch list, run the program in dbx. When the exception you are trapping occurs, the SIGFPE signal is raised and dbx stops the program. Then you can trace the call stack using the dbx where command to help find the specific line number of the program where the exception occurs (see "where Command" in the Using dbx Commands section of the Sun WorkShop online help).

Determining the Cause of the Exception

To determine the cause of the exception, use the regs -f command to display the floating point state register (FSR). Look at the accrued exception (aexc) and current exception (cexc) fields of the register, which contain bits for the following floating-point exception conditions:

- Invalid operand
- Overflow
- Underflow
- Division by zero

- Inexact result

For more information on the floating-point state register, see Version 8 (for V8) or Version 9 (for V9) of *The SPARC Architecture Manual*.For more discussion and examples, see the *Numerical Computation Guide*.

Sending a Signal in a Program

The `dbx cont` command supports the `-sig` *signal* option, which lets you resume execution of a program with the program behaving as if it had received the system signal *signal*.

For example, if a program has an interrupt handler for `SIGINT` (`^C`), you can type `^C` to stop the application and return control to `dbx`. If you issue a `cont` command by itself to continue program execution, the interrupt handler never executes. To execute the interrupt handler, send the signal, `sigint`, to the program:

```
(dbx) cont -sig int
```

The `step`, `next`, and `detach` commands accept `-sig` as well.

Automatically Handling Signals

The event management commands can also deal with signals as events. These two commands have the same effect.

```
(dbx) stop sig signal
(dbx) catch signal
```

Having the signal event is more useful if you need to associate some pre-programmed action.

```
(dbx) when sig SIGCLD {echo Got $sig $signame;}
```

In this case, make sure to first move `SIGCLD` to the ignore list.

```
(dbx) ignore SIGCLD
```

CHAPTER 15

Debugging C++

This chapter describes how `dbx` handles C++ exceptions and debugging C++ templates, including a summary of commands used when completing these tasks and examples with code samples.

This chapter is organized into the following sections:

- Using `dbx` with C++
- Exception Handling in `dbx`
- Debugging With C++ Templates

For information on compiling C++ programs, see "Debugging Optimized Code" on page 16.

Using `dbx` with C++

Although this chapter concentrates on two specific aspects of debugging C++, `dbx` allows you full functionality when debugging your C++ programs. You can:

- Find out about class and type definitions (see "Looking Up Definitions of Types and Classes" on page 42)
- Print or display inherited data members (see "Printing C++" on page 98)
- Find out dynamic information about an object pointer (see "Printing C++" on page 98)
- Debug virtual functions (see "Calling a Function" on page 54)
- Using runtime type information (see "Printing the Value of a Variable or an Expression" on page 98)
- Set breakpoints on all member functions of a class (see "Setting Breakpoints in Member Functions of the Same Class" on page 61)
- Set breakpoints on all overloaded member functions (see "Setting Breakpoints in Member Functions of Different Classes" on page 61)

- Set breakpoints on all overloaded nonmember functions (see "Setting Multiple Breakpoints in Nonmember Functions" on page 61)
- Set breakpoints on all member functions of a particular object (see "Setting Breakpoints in Objects" on page 62)
- Deal with overloaded functions/data members (see "Setting a `stop` Breakpoint in a Function" on page 58)

Exception Handling in `dbx`

A program stops running if an exception occurs. Exceptions signal programming anomalies, such as division by zero or array overflow. You can set up blocks to catch exceptions raised by expressions elsewhere in the code.

While debugging a program, `dbx` enables you to:

- Catch unhandled exceptions before stack unwinding
- Catch unexpected exceptions
- Catch specific exceptions whether handled or not before stack unwinding
- Determine where a specific exception would be caught if it occurred at a particular point in the program

If you give a `step` command after stopping at a point where an exception is thrown, control is returned at the start of the first destructor executed during stack unwinding. If you `step` out of a destructor executed during stack unwinding, control is returned at the start of the next destructor. When all destructors have been executed, a `step` command brings you to the catch block handling the throwing of the exception

Commands for Handling Exceptions

`exception [-d | +d]` Command

Use the `exception` command to display an exception's type at any time during debugging. If you use the `exception` command without an option, the type shown is determined by the setting of the `dbx` environment variable `output_dynamic_type`:

- If it is set to `on`, the derived type is shown.
- If it is set to `off` (the default), the static type is shown.

Specifying the `-d` or `+d` option overrides the setting of the environment variable:

- If you specify `-d`, the derived type is shown.
- If you specify `+d`, the static type is shown.

For more information, see "exception Command" in the Using dbx Commands section of the Sun WorkShop online help.

`intercept [-a | -x |` *typename*`]` Command

You can intercept, or catch, exceptions of a specific type before the stack has been unwound. Use the `intercept` command with no arguments to list the types that are being intercepted. Use `-a` to intercept all exceptions. Use *typename* to add a type to the intercept list. Use `-x` to exclude a particular type from being intercepted.

For example, to intercept all types except `int`, you could type:

```
(dbx) intercept -a
(dbx) intercept -x int
```

For more information, see "intercept Command" in the Using dbx Commands section of the Sun WorkShop online help.

`unintercept [-a | -x |` *typename*`]` Command

Use the `unintercept` command to remove exception types from the intercept list. Use the command with no arguments to list the types that are being intercepted (same as the `intercept` command). Use `-a` to remove all intercepted types from the list. Use *typename* to remove a type from the intercept list. Use `-x` to stop excluding a particular type from being intercepted.

For more information, see "unintercept Command" in the Using dbx Commands section of the Sun WorkShop online help.

`whocatches` *typename* Command

The `whocatches` command reports where an exception of *typename* would be caught if thrown at the current point of execution. Use this command to find out what would happen if an exception were thrown from the top frame of the stack.

The line number, function name, and frame number of the catch clause that would catch *typename* are displayed.

Examples of Exception Handling

This example demonstrates how exception handling is done in `dbx` using a sample program containing exceptions. An exception of type `int` is thrown in the function `bar` and is caught in the following catch block.

```
1  #include <stdio.h>
2
3  class c {
4      int x;
5    public:
6      c(int i) { x = i; }
7      ~c() {
8          printf("destructor for c(%d)\n", x);
9      }
10 };
```

```
11
12 void bar() {
13     c c1(3);
14     throw(99);
15 }
16
17 int main() {
18     try {
19         c c2(5);
20         bar();
21         return 0;
22     }
23     catch (int i) {
24         printf("caught exception %d\n", i);
25     }
26 }
```

The following transcript from the example program shows the exception handling features in dbx.

```
(dbx) intercept
-unhandled -unexpected
(dbx) intercept int
<dbx> intercept
-unhandled -unexpected int
(dbx) stop in bar
(2) stop in bar()
(dbx)run
Running: a.out
(process id 304)
Stopped in bar at line 13 in file “foo.cc”
   13       c c1(3);
(dbx) whocatches int
int is caught at line 24, in function main (frame number 2)
(dbx) whocatches c
dbx: no runtime type info for class c (never thrown or caught)
(dbx) cont
Exception of type int is caught at line 24, in function main
(frame number 4)
stopped in _exdbg_notify_of_throw at 0xef731494
0xef731494: _exdbg_notify_of_throw       :      jmp    %o7 + 0x8
Current function is bar
   14        throw(99);
```

```
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8         printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
stopped in c::~c at line 9 in file "foo.cc"
    9       }
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8        printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
stopped in c::~c at line 9 in file "foo.cc"
    9       )
(dbx) step
stopped in main at line 24 in file "foo.cc"
   24           printf("caught exception %d\n", i);
(dbx) step
caught exception 99
stopped in main at line 26 in file "foo.cc"
   26   }
```

Debugging With C++ Templates

`dbx` supports C++ templates. You can load programs containing class and function templates into `dbx` and invoke any of the `dbx` commands on a template that you would use on a class or function, such as:

- Setting breakpoints at class or function template instantiations (see "`stop inclass` classname Command" on page 193, "`stop infunction` name Command" on page 193, and "`stop in` function Command" on page 193)
- Printing a list of all class and function template instantiations (see "`whereis` name Command" on page 190)
- Displaying the definitions of templates and instances (see "`whatis` name Command" on page 191)
- Calling member template functions and function template instantiations (see "`call` function_name (parameters) Command" on page 194)
- Printing values of function template instantiations ("`print` Expressions" on page 194)
- Displaying the source code for function template instantiations (see "`list` Expressions" on page 195)

Template Example

The following code example shows the class template `Array` and its instantiations and the function template square and its instantiations.

```
 1  template<class C> void square(C num, C *result)
 2  {
 3      *result = num * num;
 4  }
 5
 6  template<class T> class Array
 7  {
 8  public:
 9      int getlength(void)
10      {
11          return length;
12      }
13
14      T & operator[](int i)
15      {
16          return array[i];
17      }
18
19      Array(int l)
20      {
21          length = l;
22          array = new T[length];
23      }
24
25      ~Array(void)
26      {
27          delete [] array;
28      }
29
30  private:
31      int length;
32      T *array;
33  };
34
```

```
35  int main(void)
36  {
37      int i, j = 3;
38      square(j, &i);
39
40      double d, e = 4.1;
41      square(e, &d);
42
43      Array<int> iarray(5);
44      for (i = 0; i < iarray.getlength(); ++i)
45      {
46          iarray[i] = i;
47      }
48
49      Array<double> darray(5);
50      for (i = 0; i < darray.getlength(); ++i)
51      {
52          darray[i] = i * 2.1;
53      }
54
55      return 0;
56  }
```

In the example:

- `Array` is a class template
- `square` is a function template
- `Array<int>` is a class template instantiation (template class)
- `Array<int>::getlength` is a member function of a template class
- `square(int, int*)` and `square(double, double*)` are function template instantiations (template functions)

Commands for C++ Templates

Use these commands on templates and template instantiations. Once you know the class or type definitions, you can print values, display source listings, or set breakpoints.

`whereis` *name* Command

Use the `whereis` command to print a list of all occurrences of function or class instantiations for a function or class template.

For a class template:

```
(dbx) whereis Array
member function: ‘Array<int>::Array(int)
member function: ‘Array<double>::Array(int)
class template instance: ‘Array<int>
class template instance: ‘Array<double>
class template: ‘a.out‘template_doc_2.cc‘Array
```

For a function template:

```
(dbx) whereis square
function template instance: ‘square<int>(__type_0,__type_0*)
function template instance: ‘square<double>(__type_0,__type_0*)
function template:      ‘a.out‘square
```

The `__type_0` parameter refers to the 0th template parameter. A `__type_1` would refer to the next template parameter.

For more information, see “whereis Command” in the Using dbx Commands section of the Sun WorkShop online help.

`whatis` *name* Command

Use the `whatis` command to print the definitions of function and class templates and instantiated functions and classes.

For a class template:

```
(dbx) whatis -t Array
template<class T> class Array
To get the full template declaration, try ‘whatis -t Array<int>’;
```

For the class template's constructors:

```
(dbx) whatis Array
More than one identifier 'Array'.
Select one of the following:
 0) Cancel
 1) Array<int>::Array(int)
 2) Array<double>::Array(int>
> 1
Array<int>::Array(int 1);
```

For a function template:

```
(dbx) whatis square
More than one identifier 'square'.
Select one of the following:
 0) Cancel
 1) square<int(__type_0,__type_0*)
 2) square<double>(__type_0,__type_0*)
> 2
void square<double>(double num, double *result);
```

For a class template instantiation:

```
(dbx) whatis -t Array<double>
class Array<double>; {
public:
    int Array<double>::getlength()
    double &Array<double>::operator [](int i);
    Array<double>::Array<double>(int l);
    Array<double>::~Array<double>();
private:
    int length;
    double *array;
};
```

For a function template instantiation:

```
(dbx) whatis square(int, int*)
void square(int num, int *result);
```

For more information, see "whatis Command" and "whatis Command Used With C++" in the Using dbx Commands section of the Sun WorkShop online help.

`stop inclass` *classname* Command

To stop in all member functions of a template class:

```
(dbx)stop inclass Array
(2) stop inclass Array
```

Use the `stop inclass` command to set breakpoints at all member functions of a particular template class:

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

For more information, see “stop Command” in the Using dbx Commands section of the Sun WorkShop online help.

`stop infunction` *name* Command

Use the `stop infunction` command to set breakpoints at all instances of the specified function template:

```
(dbx) stop infunction square
(9) stop infunction square
```

For more information, see “stop Command” in the Using dbx Commands section of the Sun WorkShop online help.

`stop in` *function* Command

Use the `stop in` command to set a breakpoint at a member function of a template class or at a template function.

For a member of a class template instantiation:

```
(dbx) stop in Array<int>::Array(int l)
(2) stop in Array<int>::Array(int)
```

For a function instantiation:

```
(dbx) stop in square(double, double*)
(6) stop in square(double, double*)
```

For more information, see “stop Command” in the Using dbx Commands section of the Sun WorkShop online help.

`call` *function_name* (*parameters*) Command

Use the `call` command to explicitly call a function instantiation or a member function of a class template when you are stopped in scope. If `dbx` is unable to choose the correct instance, a menu lets you choose it.

```
(dbx) call square(j,&i)
```

For more information, see “call Command” in the Using dbx Commands section of the Sun WorkShop online help.

`print` Expressions

Use the `print` command to evaluate a function instantiation or a member function of a class template:.

```
(dbx) print iarray.getlength()
iarray.getlength() = 5
```

Use `print` to evaluate the `this` pointer.

```
(dbx) whatis this
class Array<int> *this;
(dbx) print *this
*this = {
    length = 5
    array   = 0x21608
}
```

For more information, see “print Command” in the Using dbx Commands section of the Sun WorkShop online help.

`list` Expressions

Use the `list` command to print the source listing for the specified function instantiation.

```
(dbx) list square(int, int*)
```

For more information, see “list Command” in the Using dbx Commands section of the Sun WorkShop online help.

CHAPTER 16

Debugging Fortran Using dbx

This chapter introduces `dbx` features you might use with Fortran. Sample requests to `dbx` are also included to provide you with assistance when debugging Fortran code using `dbx`.

This chapter includes the following topics:

- Debugging Fortran
- Debugging Segmentation Faults
- Locating Exceptions
- Tracing Calls
- Working With Arrays
- Showing Intrinsic Functions
- Showing Complex Expressions
- Showing Logical Operators
- Viewing Fortran 95 Derived Types
- Pointer to Fortran 95 Derived Type

Debugging Fortran

The following tips and general concepts are provided to help you while debugging Fortran programs.

Current Procedure and File

During a debug session, `dbx` defines a procedure and a source file as current. Requests to set breakpoints and to print or set variables are interpreted relative to the current function and file. Thus, `stop at 5` sets different breakpoints, depending on which file is current.

Uppercase Letters

If your program has uppercase letters in any identifiers, `dbx` recognizes them. You need not provide case-sensitive or case-insensitive commands, as in some earlier versions.

FORTRAN 77, Fortran 95, and `dbx` must be in the same case-sensitive or case-insensitive mode:

- Compile and debug in case-insensitive mode without the `-U` option. The default value of the `dbx input_case_sensitive` environment variable is then `false`.

If the source has a variable named `LAST`, then in `dbx`, both the `print LAST` or `print last` commands work. FORTRAN 77, Fortran 95, and `dbx` consider `LAST` and `last` to be the same, as requested.

- Compile and debug in case-sensitive mode using `-U`. The default value of the `dbx input_case_sensitive` environment variable is then `true`.

If the source has a variable named `LAST` and one named `last`, then in `dbx`, `print LAST` works, but `print last` does not work. FORTRAN 77, Fortran 95, and `dbx` distinguish between `LAST` and `last`, as requested.

Note – File or directory names are always case-sensitive in `dbx`, even if you have set the `dbx input_case_sensitive` environment variable to `false`.

Optimized Programs

To debug optimized programs:

- Compile the main program with `-g` but without `-O`*n*.
- Compile every other routine of the program with the appropriate `-O`*n*.
- Start the execution under `dbx`.
- Use the `fix -g` *any*`.f` command on the routine you want to debug, but without `-O`*n*.
- Use the `cont` command with that routine compiled.

Main program for debugging:

`a1.f`

```
      PARAMETER ( n=2 )
      REAL twobytwo(2,2) / 4 *-1 /
      CALL mkidentity( twobytwo, n )
      PRINT *, determinant( twobytwo )
      END
```

Subroutine for debugging:

a2.f

```
    SUBROUTINE mkidentity ( array, m )
    REAL array(m,m)
    DO 90 i = 1, m
    DO 20 j = 1, m
        IF ( i .EQ. j ) THEN
        array(i,j) = 1.
        ELSE
        array(i,j) = 0.
        END IF
20  CONTINUE
90  CONTINUE
    RETURN
    END
```

Function for debugging:

a3.f

```
    REAL FUNCTION determinant ( a )
    REAL a(2,2)
    determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
    RETURN
    END
```

Sample dbx Session

The following examples use a sample program called `my_program`.

1. **Compile and link with the `-g` option.**

   You can do this in one or two steps.

   Compile and link in one step, with `-g`:

```
demo% f95 -o my_program -g a1.f a2.f a3.f
```

   Or, compile and link in separate steps:

```
demo% f95 -c -g a1.f a2.f a3.f
demo% f95 -o my_program a1.o a2.o a3.o
```

2. **Start `dbx` on the executable named `my_program`.**

```
demo% dbx my_program
Reading symbolic information…
```

3. **Set a simple breakpoint by typing `stop in` *subnam*, where *subnam* names a subroutine, function, or block data subprogram.**

   To stop at the first executable statement in a main program.

```
(dbx) stop in MAIN
(2) stop in MAIN
```

   Although `MAIN` must be all uppercase, *subnam* can be uppercase or lowercase.

4. **Type the `run` command, which runs the program in the executable files named when you started `dbx`.**

```
(dbx) run
Running: my_program
stopped in MAIN at line 3 in file "a1.f"
   3   call mkidentity( twobytwo, n )
```

   When the breakpoint is reached, `dbx` displays a message showing where it stopped—in this case, at line 3 of the `a1.f` file.

5. **To print a value, type the `print` command.**

   Print value of `n`:

```
(dbx) print n
n = 2
```

   Print the matrix `twobytwo`; the format might vary:

```
(dbx) print twobytwo
twobytwo =
   (1,1)       -1.0
   (2,1)       -1.0
   (1,2)       -1.0
   (2,2)       -1.0
```

   Print the matrix `array`:

```
(dbx) print array
dbx: "array" is not defined in the current scope
(dbx)
```

   The print fails because `array` is not defined here—only in `mkidentity`.

6. **To advance execution to the next line, type the `next` command.**

   Advance execution to the next line:

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4  print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx) quit
demo%
```

   The `next` command executes the current source line and stops at the next line. It counts subprogram calls as single statements.

   Compare the `next` command with the `step` command. The `step` command executes the next source line or the next step into a subprogram. If the next executable source statement is a subroutine or function call, then:

   - The `step` command sets a breakpoint at the first source statement of the subprogram.
   - The `next` command sets the breakpoint at the first source statement after the call, but still in the calling program.

7. **To quit `dbx`, type the `quit` command.**

```
(dbx)quit
demo%
```

Debugging Segmentation Faults

If a program gets a segmentation fault (`SIGSEGV`), it references a memory address outside of the memory available to it.

The most frequent causes for a segmentation fault are:

- An array index is outside the declared range.
- The name of an array index is misspelled.

- The calling routine has a `REAL` argument, which the called routine has as `INTEGER`.
- An array index is miscalculated.
- The calling routine has fewer arguments than required.
- A pointer is used before it has been defined.

Using `dbx` to Locate Problems

Use `dbx` to find the source code line where a segmentation fault has occurred.

Use a program to generate a segmentation fault:

```
demo% cat WhereSEGV.f
    INTEGER a(5)
    j = 2000000
    DO 9 i = 1,5
    a(j) = (i * 10)
9   CONTINUE
    PRINT *, a
    END
demo%
```

Use `dbx` to find the line number of a `dbx` segmentation fault:

```
demo% f95 -g -silent WhereSEGV.f
demo% a.out
Segmentation fault
demo% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

Locating Exceptions

If a program gets an exception, there are many possible causes. One approach to locating the problem is to find the line number in the source program where the exception occurred, and then look for clues there.

Compiling with `-ftrap=common` forces trapping on all common exceptions.

To find where an exception occurred:

```
demo% cat wh.f
                call joe(r, s)
                print *, r/s
                end
                subroutine joe(r,s)
                r = 12.
                s = 0.
                return
                end
demo% f95 -g -o wh -ftrap=common wh.f
demo% dbx wh
Reading symbolic information for wh
(dbx) catch FPE
(dbx) run
Running: wh
(process id 17970)
signal FPE (floating point divide by zero) in MAIN at line 2 in
file “wh.f”
   2                    print *, r/s
(dbx)
```

Tracing Calls

Sometimes a program stops with a core dump, and you need to know the sequence of calls that led it there. This sequence is called a *stack trace*.

The `where` command shows where in the program flow execution stopped and how execution reached this point—a *stack trace* of the called routines.

`ShowTrace.f` is a program contrived to get a core dump a few levels deep in the call sequence—to show a stack trace.

Show the sequence of calls, starting at where the execution stopped:

Note the reverse order:

`MAIN` called `calc`
`calc` called `calcb`.

Execution stopped, line 23

`calcB` called from `calc`, line 9

`calc` called from `MAIN`, line 3

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Reading symbolic information for a.out
...
(dbx) run
Running: a.out
(process id 1089)
signal SEGV (no mapping at the fault address) in calcb at
line 23 in file "ShowTrace.f"
   23                   v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v = ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a = ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx)
```

Working With Arrays

`dbx` recognizes arrays and can print them.

```
demo% dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1           DIMENSION IARR(4,4)
    2           DO 90 I = 1,4
    3                   DO 20 J = 1,4
    4                           IARR(I,J) = (I*10) + J
    5   20              CONTINUE
    6   90      CONTINUE
    7           END
(dbx)  stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out
```

```
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7           END
(dbx) print IARR
iarr =
    (1,1) 11
    (2,1) 21
    (3,1) 31
    (4,1) 41
    (1,2) 12
    (2,2) 22
    (3,2) 32
    (4,2) 42
    (1,3) 13
    (2,3) 23
    (3,3) 33
    (4,3) 43
    (1,4) 14
    (2,4) 24
    (3,4) 34
    (4,4) 44
(dbx) print IARR(2,3)
    iarr(2, 3) = 23  - Order of user-specified subscripts ok
(dbx) quit
```

For information on array slicing in Fortran, see “Array Slicing Syntax for Fortran” on page 103.

Fortran 95 Allocatable Arrays

The following example shows how to work with allocated arrays in `dbx`.

Alloc.f95

```
demo% f95 -g Alloc.f95
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM TestAllocate
    2   INTEGER n, status
    3   INTEGER, ALLOCATABLE :: buffer(:)
    4           PRINT *, 'Size?'
    5           READ *, n
    6           ALLOCATE( buffer(n), STAT=status )
    7          IF ( status /= 0 ) STOP 'cannot allocate buffer'
    8           buffer(n) = n
    9           PRINT *, buffer(n)
   10           DEALLOCATE( buffer, STAT=status)
   11   END
```

```
(dbx) stop at 6
(2) stop at "alloc.f95":6
(dbx) stop at 9
(3) stop at "alloc.f95":9
(dbx) run
Running: a.out
(process id 10749)
 Size?
1000
stopped in main at line 6 in file "alloc.f95"
    6           ALLOCATE( buffer(n), STAT=status )
(dbx) whatis buffer
integer*4 , allocatable::buffer(:)
(dbx) next
continuing
stopped in main at line 7 in file "alloc.f95"
   7          IF ( status /= 0 ) STOP 'cannot allocate buffer'
(dbx) whatis buffer
integer*4 buffer(1:1000)
(dbx) cont
stopped in main at line 9 in file "alloc.f95"
    9           PRINT *, buffer(n)
(dbx) print n
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
```

Unknown size at line 6

Known size at line 9

buffer(1000) holds 1000

```
(dbx) stop at 6
(2) stop at "alloc.f95":6
(dbx) stop at 9
(3) stop at "alloc.f95":9
(dbx) run
Running: a.out
(process id 10749)
 Size?
1000
stopped in main at line 6 in file "alloc.f95"
    6           ALLOCATE( buffer(n), STAT=status )
(dbx) whatis buffer
integer*4 , allocatable::buffer(:)
(dbx) next
continuing
stopped in main at line 7 in file "alloc.f95"
    7          IF ( status /= 0 ) STOP 'cannot allocate buffer'
(dbx) whatis buffer
integer*4 buffer(1:1000)
(dbx) cont
stopped in main at line 9 in file "alloc.f95"
    9           PRINT *, buffer(n)
(dbx) print n
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
```

Unknown size at line 6

Known size at line 9

buffer(1000) holds 1000

Showing Intrinsic Functions

dbx recognizes Fortran intrinsic functions.

To show an intrinsic function in dbx, type:

```
demo% cat ShowIntrinsic.f
    INTEGER i
    i = -2
    END
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: shi
(process id 18019)
stopped in MAIN at line 2 in file "shi.f"
    2               i = -2
(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print i
i = 0
(dbx) step
stopped in MAIN at line 3 in file "shi.f"
    3               end
(dbx) print i
i = -2
(dbx) print abs(1)
abs(i) = 2
(dbx)
```

Showing Complex Expressions

dbx also recognizes Fortran complex expressions.

To show a complex expression in dbx, type:

```
demo% cat ShowComplex.f
   COMPLEX z
    z = ( 2.0, 3.0 )
    END
demo% f95 -g -silent ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%
```

Showing Logical Operators

dbx can locate Fortran logical operators and print them.

To show logical operators in dbx, type:

```
demo% cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
demo% f95 -g ShowLogical.f
demo% dbx a.out
(dbx) list 1,9
    1           LOGICAL a, b, y, z
    2           a = .true.
    3           b = .false.
    4           y = .true.
    5           z = .false.
    6           END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx) whatis y
logical*4 y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit
demo%
```

Viewing Fortran 95 Derived Types

You can show structures—Fortran 95 derived types—with `dbx`.

```
demo% f95 -g DebStruc.f95
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM Struct ! Debug a Structure
    2      TYPE product
    3         INTEGER        id
    4         CHARACTER*16   name
    5         CHARACTER*8    model
    6         REAL           cost
    7         REAL           price
    8      END TYPE product
    9
   10      TYPE(product) :: prod1
   11
   12      prod1%id = 82
   13      prod1%name = "Coffee Cup"
   14      prod1%model = "XL"
   15      prod1%cost = 24.0
   16      prod1%price = 104.0
   17      WRITE ( *, * ) prod1%name
   18   END
(dbx) stop at 17
(2) stop at "Struct.f95":17
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f95"
   17      WRITE ( *, * ) prod1%name
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
    integer*4 id
    character*16 name
    character*8 model
    real*4 cost
    real*4 price
end type product
(dbx) n
```

```
(dbx) print prod1
prod1 = (
    id    = 82
    name = 'Coffee Cup'
    model = 'XL'
    cost = 24.0
    price = 104.0
)
```

Pointer to Fortran 95 Derived Type

You can show structures—Fortran 95 derived types—and pointers with `dbx`.

`DebStruc.f95`

Declare a derived type.

Declare `prod1` and `prod2` targets.
Declare `curr` and `prior` pointers.

Make `curr` point to `prod1`.
Make `prior` point to `prod1`.
Initialize `prior`.

Set `curr` to `prior`.
Print `name` from `curr` and `prior`.

```
demo% f95 -o debstr -g DebStruc.f95
demo% dbx debstr
(dbx) stop in main
(2) stop in main
(dbx) list 1,99
    1   PROGRAM DebStruPtr! Debug structures & pointers
    2      TYPE product
    3         INTEGER        id
    4         CHARACTER*16   name
    5         CHARACTER*8    model
    6         REAL           cost
    7         REAL           price
    8      END TYPE product
    9
   10      TYPE(product), TARGET :: prod1, prod2
   11      TYPE(product), POINTER :: curr, prior
   12
   13      curr => prod2
   14      prior => prod1
   15      prior%id = 82
   16      prior%name = "Coffee Cup"
   17      prior%model = "XL"
   18      prior%cost = 24.0
   19      prior%price = 104.0
   20      curr = prior
   21      WRITE ( *, * ) curr%name, " ", prior%name
   22   END PROGRAM DebStruPtr
(dbx) stop at 21
(1) stop at "DebStruc.f95":21
(dbx) run
Running: debstr
```

```
(process id 10972)
stopped in main at line 21 in file "DebStruc.f95"
   21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) print prod1
 prod1 = (
    id = 82
    name = "Coffee Cup"
    model = "XL"
    cost = 24.0
    price = 104.0
)
```

Above, `dbx` displays all fields of the derived type, including field names.

You can use structures—inquire about an item of an Fortran 95 derived type.

Ask about the variable

Ask about the type (`-t`)

```
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
   integer*4 id
   character*16 name
   character*8 model
   real cost
   real price
end type product
```

To print a pointer, type:

`dbx` displays the contents of a pointer, which is an address. This address can be different with every run.

```
(dbx) print prior
prior = (
    id    = 82
    name = 'Coffee Cup'
    model = 'XL'
    cost = 24.0
    price = 104.0
)
```

CHAPTER 17

Debugging at the Machine-Instruction Level

This chapter describes how to use event management and process control commands at the machine-instruction level, how to display the contents of memory at specified addresses, and how to display source lines along with their corresponding machine instructions. The `next`, `step`, `stop` and `trace` commands each support a machine-instruction level variant: `nexti`, `stepi`, `stopi`, and `tracei`. Use the `regs` command to print out the contents of machine registers or the `print` command to print out individual registers.

This chapter is organized into the following sections:

- Examining the Contents of Memory
- Stepping and Tracing at Machine-Instruction Level
- Setting Breakpoints at the Machine-Instruction Level
- Using the `adb` Command
- Using the `regs` Command

Examining the Contents of Memory

Using addresses and the `examine` or `x` command, you can examine the content of memory locations as well as print the assembly language instruction at each address. Using a command derived from `adb`(1), the assembly language debugger, you can query for:

- The *address*, using the = (equal sign) character, or,
- The *contents* stored at an address, using the / (slash) character.

You can print the assembly commands using the `dis` and `listi` commands. (See "Using the `dis` Command" on page 221 and "Using the `listi` Command" on page 221.)

Using the `examine` or `x` Command

Use the `examine` command, or its alias `x`, to display memory contents or addresses.

Use the following syntax to display the contents of memory starting at *address* for *count* items in format *fmt*. The default *addr* is the next one after the last address previously displayed. The default *count* is 1. The default *fmt* is the same as was used in the previous `examine` command, or `X` if this is the first command given.

The syntax for the `examine` command is:

```
examine [address] [/ [count] [format]]
```

To display the contents of memory from *address1* through *address2* inclusive, in format *fmt*, type:

```
examine address1, address2 [/ [format]]
```

Display the address, instead of the contents of the address in the given format by typing:

```
examine address = [format]
```

To print the value stored at the next address after the one last displayed by `examine`, type:

```
examine +/ i
```

To print the value of an expression, enter the expression as an address:

```
examine address=format
examine address=
```

Addresses

The *address* is any expression resulting in or usable as an address. The *address* may be replaced with a + (plus sign), which displays the contents of the next address in the default format.

For example, the following are valid addresses.:

`0xff99`	An absolute address
`main`	Address of a function
`main+20`	Offset from a function address
`&errno`	Address of a variable
`str`	A pointer-value variable pointing to a string

Symbolic addresses used to display memory are specified by preceding a name with an ampersand (&). Function names can be used without the ampersand; `&main` is equal to `main`. Registers are denoted by preceding a name with a dollar sign ($).

Formats

The *format* is the address display format in which `dbx` displays the results of a query. The output produced depends on the current display *format*. To change the display format, supply a different *format* code.

The default format set at the start of each `dbx` session is `X`, which displays an address or value as a 32-bit word in hexadecimal. The following memory display formats are legal.

i	Display as an assembly instruction.
d	Display as 16 bits (2 bytes) in decimal.
D	Display as 32 bits (4 bytes) in decimal.
o	Display as 16 bits (2 bytes) in octal.
O	Display as 32 bits (4 bytes) in octal.
x	Display as 16 bits (2 bytes) in hexadecimal.
X	Display as 32 bits (4 bytes) in hexadecimal. (default format)
b	Display as a byte in octal.
c	Display as a character.
w	Display as a wide character.
s	Display as a string of characters terminated by a null byte.
W	Display as a wide character.
f	Display as a single-precision floating point number.
F, g	Display as a double-precision floating point number.

E	Display as an extended-precision floating point number.
ld, lD	Display 32 bits (4 bytes) in decimal (same as D).
lo, lO	Display 32 bits (4 bytes) in octal (same as O).
lx, LX	Display 32 bits (4 bytes) in hexadecimal (same as X).
Ld, LD	Display 64 bits (8 bytes) in decimal.
Lo, LO	Display 64 bits (8 bytes) in octal .
Lx, LX	Display 64 bits (8 bytes) in hexadecimal.

Count

The *count* is a repetition count in decimal. The increment size depends on the memory display format.

Examples of Using an Address

The following examples show how to use an address with *count* and *format* options to display five successive disassembled instructions starting from the current stopping point.

For SPARC:

```
(dbx) stepi
stopped in main at 0x108bc
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
(dbx) x 0x108bc/5i
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
0x000108c0: main+0x0010: mov   0x1,%l0
0x000108c4: main+0x0014: or    %l0,%g0, %o0
0x000108c8: main+0x0018: call  0x00020b90 [unresolved PLT 8:
malloc]
0x000108cc: main+0x001c: nop
```

For Intel:

```
(dbx) x &main/5i
0x08048988: main       :  pushl  %ebp
0x08048989: main+0x0001:  movl   %esp,%ebp
0x0804898b: main+0x0003:  subl   $0x28,%esp
0x0804898e: main+0x0006:  movl   0x8048ac0,%eax
0x08048993: main+0x000b:  movl   %eax,-8(%ebp)
```

Using the `dis` Command

The `dis` command is equivalent to the `examine` command with `i` as the default display format.

Here is the syntax for the `dis` command.

```
dis [address] [address1, address2] [/count]
```

The `dis` command:

- Without arguments displays 10 instructions starting at +.
- With the *address* argument only, disassembles 10 instructions starting at *address*.
- With the *address1* and *address2* arguments, disassembles instructions from *address1* through *address2*.
- With only a *count*, displays count instructions starting at +.

Using the `listi` Command

To display source lines with their corresponding assembly instructions, use the `listi` command, which is equivalent to the command `list -i`. See the discussion of `list -i` in "Printing a Source Listing" on page 36.

For SPARC:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x0001083c: main+0x0014:  ld      [%fp + 0x48], %l0
0x00010840: main+0x0018:  add     %l0, 0x4, %l0
0x00010844: main+0x001c:  ld      [%l0], %l0
0x00010848: main+0x0020:  or      %l0, %g0, %o0
0x0001084c: main+0x0024:  call    0x000209e8 [unresolved PLT 7:
atoi]
0x00010850: main+0x0028:  nop
0x00010854: main+0x002c:  or      %o0, %g0, %l0
0x00010858: main+0x0030:  st      %l0, [%fp - 0x8]
   14       j = foo(i);
0x0001085c: main+0x0034:  ld      [%fp - 0x8], %l0
0x00010860: main+0x0038:  or      %l0, %g0, %o0
0x00010864: main+0x003c:  call    foo
0x00010868: main+0x0040:  nop
0x0001086c: main+0x0044:  or      %o0, %g0, %l0
0x00010870: main+0x0048:  st      %l0, [%fp - 0xc]
```

For Intel:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x080488fd: main+0x000d:  movl   12(%ebp),%eax
0x08048900: main+0x0010:  movl   4(%eax),%eax
0x08048903: main+0x0013:  pushl  %eax
0x08048904: main+0x0014:  call   atoi <0x8048798>
0x08048909: main+0x0019:  addl   $4,%esp
0x0804890c: main+0x001c:  movl   %eax,-8(%ebp)
   14       j = foo(i);
0x0804890f: main+0x001f:  movl   -8(%ebp),%eax
0x08048912: main+0x0022:  pushl  %eax
0x08048913: main+0x0023:  call   foo <0x80488c0>
0x08048918: main+0x0028:  addl   $4,%esp
0x0804891b: main+0x002b:  movl   %eax,-12(%ebp)
```

Stepping and Tracing at Machine-Instruction Level

Machine-instruction level commands behave the same as their source level counterparts except that they operate at the level of single instructions instead of source lines.

Single Stepping at the Machine-Instruction Level

To single step from one machine instruction to the next machine instruction, use the `nexti` command or the `stepi` command

The `nexti` command and the `stepi` command behave the same as their source-code level counterparts: the `nexti` command steps *over* functions, the `stepi` command steps into a function called by the next instruction (stopping at the first instruction in the called function). The command forms are also the same. See "next Command" and "step Command" in the Using dbx Commands section of the Sun WorkShop online help for a description.

The output from the `nexti` command and the `stepi` command differs from the corresponding source level commands in two ways:

- The output includes the *address* of the instruction at which the program is stopped (instead of the source code line number).

- The default output contains the *disassembled instruction* instead of the source code line.

For example:

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18:  call support
(dbx)
```

For more information, see "nexti Command" and "stepi Command" in the Using dbx Commands section of the Sun WorkShop online help.

Tracing at the Machine-Instruction Level

Tracing techniques at the machine-instruction level work the same as at the source code level, except you use the `tracei` command For the `tracei` command, `dbx` executes a single instruction only after each check of the address being executed or the value of the variable being traced. The `tracei` command produces automatic `stepi`-like behavior: the program advances one instruction at a time, stepping into function calls.

When you use the `tracei` command, it causes the program to stop for a moment after each instruction while `dbx` checks for the address execution or the value of the variable or expression being traced. Using the `tracei` command can slow execution considerably.

For more information on trace and its event specifications and modifiers, see "Tracing Code" on page 62 and "trace Command" in the Using dbx Commands section of the Sun WorkShop online help.

Here is the general syntax for `tracei`:

```
tracei event-specification [modifier]
```

Commonly used forms of `tracei` are:

`tracei step`	Trace each instruction.
`tracei next`	Trace each instruction, but skip over calls.
`tracei at` *address*	Trace the given code address.

For more information, see “tracei Command” in the Using dbx Commands section of the Sun WorkShop online help.

For SPARC:

```
(dbx) tracei next -in main
(dbx) cont
0x00010814: main+0x0004:  clr     %l0
0x00010818: main+0x0008:  st      %l0, [%fp - 0x8]
0x0001081c: main+0x000c:  call    foo
0x00010820: main+0x0010:  nop
0x00010824: main+0x0014:  clr     %l0
....
....
(dbx) (dbx) tracei step -in foo -if glob == 0
(dbx) cont
0x000107dc: foo+0x0004:  mov     0x2, %l1
0x000107e0: foo+0x0008:  sethi   %hi(0x20800), %l0
0x000107e4: foo+0x000c:  or      %l0, 0x1f4, %l0     ! glob
0x000107e8: foo+0x0010:  st      %l1, [%l0]
0x000107ec: foo+0x0014:  ba      foo+0x1c
....
....
```

Setting Breakpoints at the Machine-Instruction Level

To set a breakpoint at the machine-instruction level, use the `stopi` command. The command accepts any *event specification*, using the syntax:

```
stopi event-specification [modifier]
```

Commonly used forms of the `stopi` command are:

```
stopi [at address] [-if cond]
stopi in function [-if cond]
```

For more infomation, see “stopi Command” in the Using dbx Commands section of the Sun WorkShop online help.

Setting a Breakpoint at an Address

To set a breakpoint at a specific address, type:

```
(dbx) stopi at address
```

For example:

```
(dbx) nexti
stopped in hand::ungrasp at 0x12638
(dbx) stopi at &hand::ungrasp
(3) stopi at &hand::ungrasp
(dbx)
```

Using the adb Command

The adb command lets you enter commands in an adb(1) syntax. You can also enter adb mode which interprets every command as adb syntax. Most adb commands are supported.

For more information, see "adb Command" in the Using dbx Commands section of the Sun WorkShop online help.

Using the regs Command

The regs command lets you print the value of all the registers.

Here is the syntax for the regs command:

```
regs [-f][-F]
```

-f includes floating point registers (single precision). -F includes floating point registers (double precision). These are SPARC-only options.

For more information, see "regs Command" in the Using dbx Commands section of the Sun WorkShop online help.

For SPARC:

```
dbx[13] regs -F
current thread: t@1
current frame:  [1]
g0-g3    0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7    0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3    0x00000003 0x00000014 0xef7562b4 0xeffff420
o4-o7    0xef752f80 0x00000003 0xeffff3d8 0x000109b8
l0-l3    0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7    0xeffff438 0x00000001 0x00000007 0xef74df54
i0-i3    0x00000001 0xeffff4a4 0xeffff4ac 0x00020c00
i4-i7    0x00000001 0x00000000 0xeffff440 0x000108c4
y        0x00000000
psr      0x40400086
pc       0x000109c0:main+0x4    mov     0x5, %l0
npc      0x000109c4:main+0x8    st      %l0, [%fp - 0x8]
f0f1     +0.00000000000000e+00
f2f3     +0.00000000000000e+00
f4f5     +0.00000000000000e+00
f6f7     +0.00000000000000e+00
...
```

Platform-Specific Registers

The following tables list platform-specific register names for SPARC and Intel that can be used in expressions.

SPARC Register Information

The following register information is for SPARC systems.

Register	Description
`$g0 through $g7`	Global registers
`$o0 through $o7`	“out” registers
`$l0 through $l7`	“local” registers
`$i0 through $i7`	“in” registers
`$fp`	Frame pointer, equivalent to register $i6
`$sp`	Stack pointer, equivalent to register $o6

Register	Description
`$y`	Y register
`$psr`	Processor state register
`$wim`	Window invalid mask register
`$tbr`	Trap base register
`$pc`	Program counter
`$npc`	Next program counter
`$f0 through $f31`	FPU “f” registers
`$fsr`	FPU status register
`$fq`	FPU queue

The `$f0f1` `$f2f3` ... `$f30f31` pairs of floating-point registers are treated as having C "double" type (normally `$fN` registers are treated as C "float" type). These pairs can also be referred to as `$d0` ... `$d30`.

The following additional registers are available on SPARC V9 and V8+ hardware:

```
$g0g1 through $g6g7
$o0o1 through $o6o7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $$d62)
```

See the *SPARC Architecture Reference Manual* and the *Sun-4 Assembly Language Reference Manual* for more information on SPARC registers and addressing.

Intel Register Information

The following register information is for Intel systems.

Register	Description
`$gs`	Alternate data segment register
`$fs`	Alternate data segment register
`$es`	Alternate data segment register
`$ds`	Data segment register
`$edi`	Destination index register
`$esi`	Source index register

Register	Description
`$ebp`	Frame pointer
`$esp`	Stack pointer
`$ebx`	General register
`$edx`	General register
`$ecx`	General register
`$eax`	General register
`$trapno`	Exception vector number
`$err`	Error code for exception
`$eip`	Instruction pointer
`$cs`	Code segment register
`$eflags`	Flags
`$es`	Alternate data segment register
`$uesp`	User stack pointer
`$ss`	Stack segment register

Commonly used registers are also aliased to their machine independent names.

Register	Description
`$sp`	Stack pointer; equivalent of `$uesp`
`$pc`	Program counter; equivalent of `$eip`
`$fp`	Frame pointer; equivalent of `$ebp`

Registers for the 80386 lower halves (16 bits) are:

Register	Description
`$ax`	General register
`$cx`	General register
`$dx`	General register
`$bx`	General register
`$si`	Source index register

Register	Description
`$di`	Destination index register
`$ip`	Instruction pointer, lower 16 bits
`$flags`	Flags, lower 16 bits

The first four 80386 16-bit registers can be split into 8-bit parts:

Register	Description
`$al`	Lower (right) half of register `$ax`
`$ah`	Higher (left) half of register `$ax`
`$cl`	Lower (right) half of register `$cx`
`$ch`	Higher (left) half of register `$cx`
`$dl`	Lower (right) half of register `$dx`
`$dh`	Higher (left) half of register `$dx`
`$bl`	Lower (right) half of register `$bx`
`$bh`	Higher (left) half of register `$bx`

Registers for the 80387 are:

register	Description
`$fctrl`	Control register
`$fstat`	Status register
`$ftag`	Tag register
`$fip`	Instruction pointer offset
`$fcs`	Code segment selector
`$fopoff`	Operand pointer offset
`$fopsel`	Operand pointer selector
`$st0 through $st7`	Data registers

CHAPTER 18

Using dbx With the Korn Shell

The `dbx` command language is based on the syntax of the Korn Shell (ksh 88), including I/O redirection, loops, built-in arithmetic, history, and command-line editing. This chapter lists the differences between ksh-88 and `dbx` command language.

If no `dbx` initialization file is located on startup, `dbx` assumes ksh mode.

This chapter is organized into the following sections:

- ksh-88 Features Not Implemented
- Extensions to ksh-88
- Renamed Commands

ksh-88 Features Not Implemented

The following features of ksh-88 are not implemented in `dbx`:

- `set -A` *name* for assigning values to array *name*
- `set -o` particular options: `allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset -l -u -L -R -H` attributes
- backquote (`` `…` ``) for command substitution (use `$(...)` instead)
- [[*expr*]] compound command for expression evaluation
- `@(pattern[|pattern] …)` extended pattern matching
- co-processes (command or pipeline running in the background that communicates with your program)

Extensions to ksh-88

`dbx` adds the following features as extensions:

- `$[ p -> flags ]` language expression
- `typeset -q` enables special quoting for user-defined functions
- csh-like `history` and `alias` arguments
- `set +o path` disables path searching
- `0xabcd` C syntax for octal and hexadecimal numbers
- `bind` to change Emacs-mode bindings
- `set -o hashall`
- `set -o ignore suspend`
- `print -e` and `read -e` (opposite of `-r`, raw)
- built-in `dbx` commands

Renamed Commands

Particular `dbx` commands have been renamed to avoid conflicts with ksh commands.

- The `dbx print` command retains the name `print`; the ksh `print` command has been renamed `kprint`.
- The ksh `kill` command has been merged with the `dbx kill` command.
- The `alias` command is the ksh `alias`, unless in `dbx` compatibility mode.
- *addr/fmt* is now `examine` *addr/fmt*.
- `/`*pattern* is now `search` *pattern*
- `?`*pattern* is now `bsearch` *pattern*.

CHAPTER 19

Debugging Shared Libraries

dbx provides full debugging support for programs that use dynamically-linked, shared libraries, provided that the libraries are compiled using the -g option.

This chapter is organized into the following sections:

- Dynamic Linker
- Debugging Support for Preloaded Shared Objects
- Fix and Continue
- Setting a Breakpoint in a Dynamically Linked Library

Dynamic Linker

The dynamic linker, also known as rtld, Runtime ld, or ld.so, arranges to bring shared objects (load objects) into an executing application. There are two primary areas where rtld is active:

- Program startup – At program startup, rtld runs first and dynamically loads all shared objects specified at link time. These are *preloaded* shared objects and may include libc.so, libC.so, or libX.so. Use ldd(1) to find out which shared objects a program will load.
- Application requests – The application uses the function calls dlopen(3) and dlclose(3) to dynamically load and unload shared objects or executables.

dbx uses the term *loadobject* to refer to a shared object (.so) or executable (a.out).

Link Map

The dynamic linker maintains a list of all loaded objects in a list called a *link map*, which is maintained in the memory of the program being debugged, and is indirectly accessed through librtld_db.so, a special system library for use by debuggers.

Startup Sequence and `.init` Sections

A `.init` section is a piece of code belonging to a shared object that is executed when the shared object is loaded. For example, the `.init` section is used by the C++ runtime system to call all static initializers in a `.so`.

The dynamic linker first maps in all the shared objects, putting them on the link map. Then, the dynamic linker traverses the link map and executes the `.init` section for each shared object. The `syncrtld` event occurs between these two phases.

Procedure Linkage Tables

Procedure linkage tables (PLTs) are structures used by the `rtld` to facilitate calls across shared object boundaries. For instance, the call to `printf` goes through this indirect table. The details of how this is done can be found in the generic and processor specific SVR4 ABI reference manuals.

For `dbx` to handle `step` and `next` commands across PLTs, it has to keep track of the PLT table of each load object. The table information is acquired at the same time as the `rtld` handshake.

Debugging Support for Preloaded Shared Objects

To put breakpoints in preloaded shared objects, `dbx` must have the addresses of the routines. For `dbx` to have the addresess of the routines, it must have the shared object base address. Doing something as simple as the following requires special consideration by `dbx`.

```
stop in printf
run
```

Whenever you load a new program, `dbx` automatically executes the program up to the point where `rtld` has completed construction of the link map. `dbx` then reads the link map and stores the base addresses. After that, the process is killed and you see the prompt. These `dbx` tasks are conducted silently.

At this point, the symbol table for `libc.so` is available as well as its base load address. Therefore, the address of `printf` is known.

The activity of dbx waiting for rtld to construct the link map and accessing the head of the link map is known as the rtld handshake. The event syncrtld occurs when rtld is done with the link map and dbx has read all of the symbol tables.

Fix and Continue

Using fix and continue with shared object sloaded with dlopen() requires a change in how they are opened for fix and continue to work correctly. Use mode RTLD_NOW|RTLD_GLOBAL or RTLD_LAZY|RTLD_GLOBAL.

Setting a Breakpoint in a Dynamically Linked Library

dbx automatically detects that a dlopen or a dlclose has occurred and loads the symbol table of the loaded object. Once a shared object has been loaded with dlopen() you can place breakpoints in it and debug it as you would any part of your program.

If a shared object is unloaded using dlclose(), dbx remembers the breakpoints placed in it and replaces them if the shared object is again loaded with dlopen(), even if the application is run again. (Versions of dbx prior to 5.0 would instead mark the breakpoint as '(defunct)', and it had to be deleted and replaced by the user.)

However, you do not need to wait for the loading of a shared object with dlopen() to place a breakpoint in it, or to navigate its functions and source code. If you know the name of the shared object that the program being debugged will be loading with dlopen(), you can arrange for dbx to preload its symbol table into dbx by using:

```
loadobjects -p /usr/java1.1/lib/libjava_g.so
```

You can now navigate the modules and functions in this loadobject and place breakpoints in it before it has ever been loaded with dlopen(). Once it is loaded, dbx automatically places the breakpoints.

Setting a breakpoint in a dynamically linked library is subject to the following limitations:

- You cannot set a breakpoint in a "filter" library loaded with dlopen() until the first function in it is called.

- When a library is loaded by `dlopen()`, an initialization routine named `_init()` is called. This routine might call other routines in the library. `dbx` cannot place breakpoints in the loaded library until after this initialization is completed. In specific terms, this means you cannot have `dbx` stop at `_init()` in a library loaded by `dlopen`.

APPENDIX **A**

Modifying a Program State

This appendix focuses on `dbx` usage and commands that change your program or change the behavior of your programwhen you run it under `dbx`, as compared to running it without `dbx`. It is important to understand which commands might make modifications to your program.

The chapter is divided into the following sections:

- Impacts of Running a Program Under `dbx`
- Commands That Alter the State of the Program

Impacts of Running a Program Under `dbx`

Your application might behave differently when run under `dbx`. Although `dbx` strives to minimize its impact on the program being debugged, you should be aware of the following:

- You might have forgotten to take out a `-C` or disable RTC. Having the RTC support library `librtc.so` loaded into a program can cause the program to behave differently.
- Your `dbx` initialization scripts might have some environment variables set that you've forgotten about. The stack base starts at a different address when running under `dbx`. This is also different based on your environment and the contents of `argv[]`, forcing local variables to be allocated differently. If they're not initialized, they will get different random numbers. This problem can be detected using runtime checking.
- The program does not initialize memory allocated with `malloc()` before use; a situation similar to the previous one. This problem can be detected using runtime checking.
- `dbx` has to catch LWP creation and `dlopen` events, which might affect timing-sensitive multithreaded applications.

- `dbx` does context switching on signals, so if your application makes heavy use of signals, things might work differently.
- Your program might be expecting that `mmap()` always returns the same base address for mapped segments. Running under `dbx` perturbs the address space sufficiently to make it unlikely that `mmap()` returns the same address as when the program is run without `dbx`. To determine if this is a problem, look at all uses of `mmap()` and ensure that the address returned is used by the program, rather than a hard-coded address.
- If the program is multithreaded, it might contain data races or be otherwise dependent upon thread scheduling. Running under `dbx` perturbs thread scheduling and may cause the program to execute threads in a different order than normal. To detect such conditions, use `lock_lint`.

Otherwise, determine whether running with `adb` or `truss` causes the same problems.

To minimize perturbations imposed by `dbx`, try attaching to the application while it is running in its natural environment.

Commands That Alter the State of the Program

`assign` Command

The `assign` command assigns a value of the *expression* to *variable*. Using it in `dbx` permanently alters the value of *var*.

```
assign variable = expression
```

`pop` Command

The `pop` command pops a frame or frames from the stack:

`pop`	Pop current frame.
`pop` *number*	Pop *number* frames.
`pop -f` *number*	Pop frames until specified frame *number*.

Any calls popped are re-executed upon resumption, which might result in unwanted program changes. `pop` also calls destructors for objects local to the popped functions.

`call` Command

When you use the `call` command in `dbx`, you call a procedure and the procedure performs as specified:

```
call proc([params])
```

The procedure could modify something in your program. `dbx` is making the call as if you had written it into your program source.

`print` Command

To print the value of the expression(s), type:

```
print expression, ...
```

If an expression has a function call, the same considerations apply as with the `call` command. With C++, you should also be careful of unexpected side effects caused by overloaded operators.

`when` Command

The `when` command has a general syntax as follows:

```
when event-specification [modifier] {command ... ;}
```

When the event occurs, the *command*s are executed.

When you get to a line or to a procedure, a command is performed. Depending upon which command is issued, this could alter your program state.

`fix` Command

You can use the `fix` command to make immediate changes to your program:

```
fix
```

Although is a very useful tool, the `fix` command recompiles modified source files and dynamically links the modified functions into the application.

Make sure to check the restrictions for fix and continue. See Chapter 11.

`cont at` Command

The `cont at` command alters the order in which the program runs. Execution is continued at line *line. id* is required if the program is multithreaded.

```
cont at line id
```

This could change the outcome of the program.

Index

SYMBOLS

:: (double-colon) C++ operator, 37

A

access checking, 116
adb command, 225
adb mode, 225
address
  display format, 219
  examining contents at, 217
adjusting default dbx settings, 23
alias command, 15
allow_critical_exclusion environment variable, 28
analyzing visualized data, 155
aout_cache_size environment variable, 28
array expressions, 147
array_bounds_check environment variable, 28
arrays
  automatic updating of displays, 150
  bounds, exceeding, 202
  changing perspective depth of displays, 151
  changing the display of, 151
  Contour graph type
    displaying in color, 154
    displaying in lines, 154
  evaluating, 102
  Fortran, 205
  Fortran 95 allocatable, 207
  graphing, 149
  preparing to graph, 149
  rotating display of, 151
  single -dimensional, how graphed, 147
  slicing, 102, 106
    graph examples, 148
    syntax for C and C++, 102
    syntax for Fortran, 103
  striding, 102, 106
  Surface graph type, 152
    shading choices for, 153
    texture choices for, 153
  syntax for slicing, striding, 102
  two-dimensional, how graphed, 147
  ways to graph, 149
  with a range of values, graph examples, 148
assembly language debugging, 217
assign command, 101, 165, 166, 238
assigning a value to a variable, 101, 238
attached process, using runtime checking on, 133
attaching
  dbx to a running child process, 175
  dbx to a running process, 12, 50
    when dbx is not already running, 51
Auto-Read facility
  and .o files, 45
  and archive files, 45
  and executable file, 46
  default behavior, 45

B

backquote operator, 37
`bcheck` command, 136
  examples, 137
  syntax, 136
block local operator, 38
breakpoints
  conditional, setting, 64
  deleting, using handler ID, 64
  enabling after event occurs, 89
  event efficiency, 68
  event specifications, 72
  In Function, 58, 62
  In Member, 61
  In Object, 62
  listing, 64
  multiple, setting in nonmember functions, 61
  On Condition, 66
  On Modify, 65
  overview, 57
  restrictions on, 60
  setting
    at a line, 58
    at a member function of a template class or at a template function, 193
    at all instances of a function template, 193
    at an address, 225
    at class template instantiations, 188, 193
    at function template instantiations, 188, 193
    conditional breaks, 65
    filters, 66
    in a function, 58
    in dynamically linked library, 60, 235
    in member functions of different classes, 61
    in member functions of the same class, 61
    in objects, 62
    in preloaded shared objects, 234
    machine level, 224
    multiple breaks in C++ code, 61
  `stop` type, 57
  `trace` type, 57
  watchpoints, 64
  `when` type, 57
    setting at a line, 60
`button` command, 26

C

C array example program, 160
C++
  ambiguous or overloaded functions, 35
  backquote operator, 37
  class
    declarations, looking up, 41
    definition, looking up, 42
    displaying all the data members directly defined by, 99
    displaying all the data members inherited from, 99
    printing the declaration of, 42
    seeing inherited members, 44
    viewing, 41
  compiling with the `-g` option, 15
  compiling with the `-g0` option, 15
  double-colon scope resolution operator, 37
  exception handling, 184
  function template instantiations, listing, 41
  inherited members, 44
  mangled names, 38
  object pointer types, 98
  printing, 98
  setting multiple breakpoints, 60, 61
  template debugging, 188
  template definitions
    displaying, 41
    fixing, 167
  tracing member functions, 63
  unnamed arguments, 99
  using dbx with, 183
`call` command, 54, 194, 239
call stack, 91
  deleting
    all frame filters, 95
    frames, 94
  finding your place on, 91
  hiding frames, 94
  moving
    down, 93
    to a specific frame in, 93
    up, 92
  popping, 94, 164, 239
    one frame of, 166
  removing the stopped in function from, 94
  walking, 36, 92

calling
a function, 54
a function instantiation or a member function of a class template, 194
a procedure, 239
member template functions, 188
case sensitivity, Fortran, 198
catch blocks, 184
`catch` command, 179, 180
catch signal list, 179
catching exceptions of a specific type, 185
`cfront_demangling` environment variable, 28
changing
a function not yet called, 164
an executed function, 164
array display perspective, 151
default signal lists, 179
function currently being executed, 164
function presently on the stack, 164
number of data value ranges shown in Contour graph type, 154
variables after fixing, 165
`check` command, 111, 112
checkpoints, saving a series of debugging runs as, 20
child process
attaching `dbx` to, 175
debugging, 175
interacting with events, 176
using runtime checking on, 129
choosing among multiple occurrences of a symbol, 35
class template instantiations, printing a list of, 188, 190
classes
displaying all the data members directly defined by, 99
displaying all the data members inherited from, 99
looking up declarations of, 41
looking up definitions of, 42
printing the declarations of, 42
seeing inherited members, 44
viewing, 41
clearing event handlers, 63
code compiled without `-g` option, 16
commands
`adb`, 225
`alias`, 15
`assign`, 101, 165, 166, 238
`bcheck`, 136
`button`, 26
`call`, 54, 194, 239
`catch`, 179, 180
`check`, 111, 112
`cont`, 52, 53, 112, 163, 164, 166, 172, 240
limitations for files compiled without debugging information, 162
`dbxenv`, 14, 27
`debug`, 175
`detach`, 18, 51
`dis`, 221
`display`, 100
`down`, 93
entering in `adb(1)` syntax, 225
`examine`, 218
`exception`, 184
`file`, 34
`fix`, 162, 163, 240
effects of, 163
limitations for files compiled without debugging information, 162
`frame`, 93
`func`, 35
`handler`, 71
`hide`, 94
`ignore`, 178, 179
`intercept`, 185
`kill`, 18, 120
`list`, 36, 195
`listi`, 221
`lwps`, 173
`modify`, 66
`module`, 47
`modules`, 47
`nexti`, 222
`pathmap`, 14, 33, 163
`pop`, 94, 166, 239
`print`, 98, 100, 102, 103, 194, 239
process control, 49
`regs`, 225
`replay`, 18, 21
`restore`, 18, 21
`run`, 49
`save`, 18
`showblock`, 111

showleaks, 120, 123, 124, 127
showmemuse, 124
step, 184
stepi, 222
stop, 193
stop at, 58
stop change, 65
stop inclass, 61
stop inmember, 61
stop modify, 65
stopi, 224
suppress, 113, 126, 128
that alter the state of your program, 238
thread, 171
threads, 172
trace, 63
tracei, 223
uncheck, 112
undisplay, 101
unhide, 95
unintercept, 185
unsuppress, 126, 128
up, 92
vitem, 150
whatis, 41, 42, 99, 191
when, 60, 70, 240
when infunction, 62
where, 91, 204
whereis, 39, 97, 190
which, 35, 40, 97
whocatches, 185
x, 218
comparing
data graphs
at different point in a program, 157
from different runs of the same program, 157
different views of the same data, 155
compiling
optimized code, 16
with the -g option, 15
with the -O option, 15
cont command, 52, 53, 112, 163, 164, 166, 172, 240
limitations for files compiled without debugging information, 162
continuing execution of a program, 52
after fixing, 163
at a specified line, 53, 240
Contour graph type for array displays
changing the number of data value ranges shown, 154
displaying in color, 154
displaying in lines, 154
controlling the speed of a trace, 63
core file debugging, 12
creating
a .dbxrc file, 24
event handlers, 70
current procedure and file, 197
custom buttons, storing, 26
customizing
array display, 151
dbx, 23
in Sun WorkShop, 25

D

data
comparing different views of, 155
comparing graphs at different points in a program, 157
updating graphs of automatically, 156
visualized, analyzing, 155
Data Graph window, 147
data member, printing, 42
data visualization, 147
dbx environment variables, 28
allow_critical_exclusion, 28
and the Korn shell, 25
aout_cache_size, 28
array_bounds_check, 28
cfront_demangling, 28
delay_xs, 28, 46
disassembler_version, 28
fix_verbose, 28
follow_fork_inherit, 28, 176
follow_fork_mode, 28, 129, 176
follow_fork_mode_inner, 29
input_case_sensitive, 29, 198
language_mode, 29
locache_enable, 29
mt_scalable, 29
output_auto_flush, 29
output_base, 29

output_derived_type, 99
output_dynamic_type, 29, 184
output_inherited_members, 29
output_list_size, 29
output_log_file_name, 29
output_max_string_length, 29
output_pretty_print, 29
output_short_file_name, 30
overload_function, 30
overload_operator, 30
pop_auto_destruct, 30
proc_exclusive_attach, 30
rtc_auto_continue, 30, 112, 138
rtc_auto_suppress, 30, 127
rtc_biu_at_exit, 30, 124
rtc_error_limit, 30, 127
rtc_error_log_file_name, 30, 113, 138
rtc_error_stack, 30
rtc_inherit, 30
rtc_mel_at_exit, 30
rtc_use_traps, 31
run_autostart, 31
run_io, 31
run_pty, 31
run_quick, 31
run_savetty, 31
run_setpgrp, 31
scope_global_enums, 31
scope_look_aside, 31
session_log_file_name, 31
setting with the dbxenv command, 27
stack_find_source, 32
stack_max_size, 32
stack_verbose, 32
step_events, 32, 68
step_granularity, 32, 52
suppress_startup_message, 32
symbol_info_compression, 32
trace_speed, 32, 63
dbx, starting, 11
with core file name, 12
with *process_ID* only, 12
dbxenv command, 14, 27
.dbxrc file, 23, 26
creating, 24
sample, 24
debug command, 175
debugging
assembly language, 217
child processes, 175
code compiled without -g option, 16
core file, 12
machine-instruction level, 217, 222
multithreaded programs, 169
optimized code, 16
optimized programs, 198
debugging information
for a module, reading in, 47
for all modules, reading in, 47
debugging options
maintaining a unified set of, 25
maintaining two sets of, 26
debugging run
saved
replaying, 21
restoring, 20
saving, 18
declarations, looking up (displaying), 41
delay_xs environment variable, 28, 46
deleting
all call stack frame filters, 95
call stack frames, 94
specific breakpoints using handler IDs, 64
dereferencing a pointer, 100
detach command, 18, 51
detaching a process from dbx, 18, 51
determining
cause of floating point exception (FPE), 180
location of floating point exception (FPE), 180
number of instructions executed, 88
number of lines executed, 88
the granularity of source line stepping, 52
which symbol dbx uses, 40
differences between Korn shell and dbx commands, 231
dis command, 221
disassembler_version environment variable, 28
display command, 100
displaying
all the data members directly defined by a class, 99
all the data members inherited from a base class, 99

an unnamed function argument, 100
declarations, 41
definitions of templates and instances, 188
inherited members, 43
source code for function template instantiations, 188
symbols, occurrences of, 39
template definitions, 41
the definitions of templates and instances, 191
type of an exception, 184
variable type, 42
variables and expressions, 100
`dlopen()`
restrictions on breakpoints, 60
setting a breakpoint, 60
`down` command, 93
dynamic linker, 233

E

enabling a breakpoint after an event occurs, 89
error suppression, 126, 127
default, 128
examples, 127
types, 126
establishing a new mapping from directory to directory, 14, 34
evaluating
a function instantiation or a member function of a class template, 194
an unnamed function argument, 100
arrays, 102
event counters, 71
event handlers
clearing, 63
creating, 70
listing, 63
manipulating, 71
setting, examples, 87
event specifications, 69, 70, 72, 224
for breakpoint events, 72
for execution progress events, 77
for other types of events, 79
for system events, 75
for watchpoint events, 73
keywords, defined, 72
modifiers, 81
setting, 72
using predefined variables, 84
events
ambiguity, 83
child process interaction with, 176
parsing, 83
watchpoints, 64
event-specific variables, 85
`examine` command, 218
examining the contents of memory, 217
`exception` command, 184
exception handling, 184
examples, 186
exceptions
in Fortran programs, locating, 204
of a specific type, catching, 185
removing types from intercept list, 185
reporting where type would be caught, 185
type of, displaying, 184
`exec` function, following, 176
execution progress event specifications, 77
expressions
complex, Fortran, 211
displaying, 100
monitoring changes, 100
monitoring the value of, 100
printing the value of, 98, 239
turning off the display of, 101

F

`fflush(stdout)`, after dbx calls, 54
field type
displaying, 42
printing, 42
`file` command, 34
files
archive, and Auto-Read facility, 45
finding, 14
location of, 33
qualifying name, 36
visiting, 34
finding
object files, 14
source files, 14
your place on the call stack, 91

fix and continue, 161
  how it operates, 162
  modifying source code using, 162
  restrictions, 162
  using with runtime checking, 134
  using with shared objects, 235
`fix` command, 162, 163, 240
  effects of, 163
  limitations for files compiled without debugging information, 162
`fix_verbose` environment variable, 28
fixing
  C++ template definitions, 167
  shared objects, 162
  your program, 163, 240
floating point exception (FPE)
  catching, 90
  determining cause of, 180
  determining location of, 180
`follow_fork_inherit` environment variable, 28, 176
`follow_fork_mode` environment variable, 28, 129, 176
`follow_fork_mode_inner` environment variable, 29
following
  the `exec` function, 176
  the `fork` function, 176
`fork` function, following, 176
Fortran
  allocatable arrays, 207
  array example program, 159
  array slicing syntax for, 103
  case sensitivity, 198
  complex expressions, 211
  derived types, 213
  intrinsic functions, 210
  logical operators, 212
  structures, 213
FPE signal, trapping, 179
`frame` command, 93
`func` command, 35
function argument, unnamed
  displaying, 100
  evaluating, 100
function template instantiations
  displaying the source code for, 188
  printing a list of, 188, 190
  printing the values of, 188
functions
  ambiguous or overloaded, 35
  calling, 54
  currently being executed, changing, 164
  executed, changing, 164
  instantiation
    calling, 194
    evaluating, 194
    printing source listing for, 195
  intrinsic, Fortran, 210
  looking up definitions of, 41
  member of a class template, calling, 194
  member of class template, evaluating, 194
  not yet called, changing, 164
  obtaining names assigned by the compiler, 99
  presently on the stack, changing, 164
  qualifying name, 36
  setting breakpoints in, 58
  setting breakpoints in C++ code, 61
  visiting, 35

G

`-g` compiler option, 15
graphing an array, 149
  from `dbx` command line, 150

H

`handler` command, 71
handler id, defined, 70
handlers, 69
  creating, 70
  enabling while within a function, 87
header file, modifying, 166
`hide` command, 94
hiding call stack frames, 94

I

`ignore` command, 178, 179
ignore signal list, 179
In Function breakpoint, 58, 62
In Member breakpoint, 61
In Object breakpoint, 62
inherited members
  displaying, 43
  seeing, 44
`input_case_sensitive` environment variable, 29, 198
instances, displaying the definitions of, 188, 191
Intel registers, 227
`intercept` command, 185

K

`kill` command, 18, 120
killing
  program, 18
  program only, 18
Korn shell
  differences from `dbx`, 231
  extensions, 232
  features not implemented, 231
  renamed commands, 232

L

`language_mode` environment variable, 29
LD_PRELOAD, 133
libraries
  dynamically linked, setting breakpoints in, 60
  shared, compiling for dbx, 17
`librtc.so`, preloading, 133
`librtld_db.so`, 233
`libthread.so`, 169
`libthread_db.so`, 169
link map, 233
linker names, 38
`list` command, 36, 195
`listi` command, 221
listing
  all program modules that contain debugging information, 48
  breakpoints, 64
  C++ function template instantiations, 41
  debugging information for modules, 47
  event handlers, 63
  names of all program modules, 48
  names of modules containing debugging information that have already been read into dbx, 47
  signals currently being ignored, 179
  signals currently being trapped, 179
  traces, 64
loadobject, 233
`locache_enable` environment variable, 29
locating
  object files, 14
  source files, 14
looking up
  definitions of classes, 42
  definitions of functions, 41
  definitions of members, 41
  definitions of types, 42
  definitions of variables, 41
  the `this` pointer, 42
LWPs (lightweight processes), 169
  information displayed for, 173
  showing information about, 173
`lwps` command, 173

M

machine-instruction level
  address, setting breakpoint at, 225
  debugging, 217
  Intel registers, 227
  printing the value of all the registers, 225
  setting breakpoint at address, 224
  single stepping, 222
  SPARC registers, 226
  tracing, 223
maintaining
  a unified set of debugging options, 25
  two sets of debugging options, 26
manipulating event handlers, 71

member functions
printing, 41
setting multiple breakpoints in, 60
tracing, 63
member template functions, 188
members
declarations, looking up, 41
looking up declarations of, 41
looking up definitions of, 41
viewing, 41
memory
address display formats, 219
display modes, 217
examining contents at address, 217
states, 116
memory access
checking, 116
turning on, 111, 112
error report, 117
errors, 117, 140
memory leak
checking, 118, 120
turning on, 111, 112
errors, 119, 144
fixing, 124
report, 121
memory use checking, 124
turning on, 111, 112
`modify` command, 66
modifying a header file, 166
`module` command, 47
modules
all program, listing, 48
containing debugging information that have already been read into dbx, listing, 47
current, printing the name of, 47
listing debugging information for, 47
that contain debugging information, listing, 48
`modules` command, 47
monitoring the value of an expression, 100
moving
down the call stack, 93
to a specific frame in the call stack, 93
up the call stack, 92
`mt_scalable` environment variable, 29
multithreaded programs, debugging, 169

N

`nexti` command, 222

O

object pointer types, 98
obtaining the function name assigned by the compiler, 99
On Condition breakpoint, 66
On Modify breakpoint, 65
operators
backquote, 37
block local, 38
C++ double colon scope resolution, 37
optimized code
compiling, 16
debugging, 16
`output_auto_flush` environment variable, 29
`output_base` environment variable, 29
`output_derived_type` environment variable, 99
`output_dynamic_type` environment variable, 29, 184
`output_inherited_members` environment variable, 29
`output_list_size` environment variable, 29
`output_log_file_name` environment variable, 29
`output_max_string_length` environment variable, 29
`output_pretty_print` environment variable, 29
`output_short_file_name` environment variable, 30
`overload_function` environment variable, 30
`overload_operator` environment variable, 30

P

`pathmap` command, 14, 33, 163
pointers
dereferencing, 100
printing, 216
`pop` command, 94, 166, 239

pop_auto_destruct environment variable, 30
popping
one frame of the call stack, 166
the call stack, 94, 164, 239
predefined variables for event specification, 84
preloading librtc.so, 133
print command, 98, 100, 102, 103, 194, 239
printing
a list of all class and function template instantiations, 188, 190
a list of occurrences of a symbol, 39
a pointer, 216
a source listing, 36
arrays, 102
data member, 42
field type, 42
list of all known threads, 172
list of threads normally not printed (zombies), 172
member functions, 41
the declaration of a type or C++ class, 42
the name of the current module, 47
the source listing for the specified function instantiation, 195
the value of a variable or expression, 98
the value of all the machine-level registers, 225
the value of an expression, 239
values of function template instantiations, 188
variable type, 42
proc_exclusive_attach environment variable, 30
procedure linkage tables, 234
procedure, calling, 239
process
attached, using runtime checking on, 133
child
attaching dbx to, 175
using runtime checking on, 129
detaching from dbx, 18, 51
running, attaching dbx to, 50, 51
stopping execution, 17
stopping with Ctrl+C, 55
process control commands, definition, 49
program
continuing execution of, 52
after fixing, 163
at a specified line, 240
fixing, 163, 240
killing, 18
multithreaded
debugging, 169
resuming execution of, 172
optimized, debugging, 198
resuming execution of at a specific line, 53
running, 49
under dbx, impacts of, 237
with runtime checking turned on, 112
single stepping through, 52
status, checking, 89
stepping through, 51
stopping execution
if a conditional statement evaluates to true, 66
if the value of a specified variable has changed, 65
when the contents of an address is written to, 64
stripped, 17

Q

qualifying symbol names, 36
quitting a dbx session, 17

R

reading in
debugging information for a module, 47
debugging information for all modules, 47
registers
Intel, 227
printing the value of, 225
SPARC, 226
regs command, 225
removing
exception types from intercept list, 185
the stopped in function from the call stack, 94
replay command, 18, 21
replaying a saved debugging run, 21
reporting where an exception type would be caught, 185
resetting application files for replay, 89
restore command, 18, 21
restoring a saved debugging run, 20

resuming
execution of a multithreaded program, 172
program execution at a specific line, 53
rotating array display, 151
rtc_auto_continue environment variable, 30, 112, 138
rtc_auto_suppress environment variable, 30, 127
rtc_biu_at_exit environment variable, 30
rtc_error_limit environment variable, 30, 127
rtc_error_log_file_name environment variable, 30, 113, 138
rtc_error_stack environment variable, 30
rtc_inherit environment variable, 30
rtc_mel_at_exit environment variable, 30
rtc_use_traps environment variable, 31
rtld, 233
run command, 49
run_autostart environment variable, 31
run_io environment variable, 31
run_pty environment variable, 31
run_quick environment variable, 31
run_savetty environment variable, 31
run_setpgrp environment variable, 31
running a program, 49
in dbx without arguments, 49
with runtime checking turned on, 112
runtime checking
8 megabyte limit, 138
a child process, 129
access checking, 116
an attached process, 133
application programming interface, 136
error suppression, 126
errors, 140
fixing memory leaks, 124
limitations, 111
memory access
checking, 116
error report, 117
errors, 117, 140
memory leak
checking, 118, 120
error report, 121
errors, 119, 144
memory use checking, 124
possible leaks, 119
requirements, 110
suppressing errors, 126
default, 128
examples, 127
suppression of last error, 127
troubleshooting tips, 138
turning off, 112
types of error suppression, 126
using fix and continue with, 134
using in batch mode, 136
directly from dbx, 138
when to use, 110

S

sample .dbxrc file, 24
save command, 18
saving
debugging run to a file, 18, 20
series of debugging runs as checkpoints, 20
scope resolution operators, 36
scope resolution search path, 38
scope_global_enums environment variable, 31
scope_look_aside environment variable, 31
search order for symbols, 38
segmentation fault
finding line number of, 203
Fortran, causes, 202
generating, 203
selecting from a list of C++ ambiguous function names, 35
session, dbx
quitting, 17
starting, 11
session_log_file_name environment variable, 31
setting
a trace, 63
breakpoint filters, 66
breakpoints
at a member function of a template class or at a template function, 193
at all instances of a function template, 193
in member functions of different classes, 61
in member functions of the same class, 61

in objects, 62
dbx environment variables with the dbxenv command, 27
multiple breakpoints in nonmember functions, 61
shared libraries, compiling for dbx, 17
shared objects
debugging support, 234
setting breakpoints in, 234
using fix and continue with, 235
shared objects, fixing, 162
showblock command, 111
showleaks command, 120, 123, 124, 127
showmemuse command, 124
signals
cancelling, 178
catching, 179
changing default lists, 179
forwarding, 178
FPE, trapping, 179
handling automatically, 181
ignoring, 179
listing those currently being ignored, 179
listing those currently being trapped, 179
sending in a program, 181
single stepping
at the machine-instruction level, 222
through a program, 52
slicing
arrays, 106
C and C++ arrays, 102
Fortran arrays, 103
Solaris versions supported, 1
source listing, printing, 36
SPARC registers, 226
specifying, 147
stack trace, 204
stack_find_source environment variable, 32
stack_max_size environment variable, 32
stack_verbose environment variable, 32
step command, 184
step_events environment variable, 32, 68
step_granularity environment variable, 32, 52
stepi command, 222
stepping through a program, 51
stop at command, 58
stop change command, 65
stop command, 193
stop inclass command, 61
stop inmember command, 61
stop modify command, 65
stopi command, 224
stopping
a process with Ctrl+C, 55
in all member functions of a template class, 193
process execution, 17
program execution
if a conditional statement evaluates to true, 66
if the value of a specified variable has changed, 65
when the contents of an address is written to, 64
storing custom buttons, 26
striding across slices of arrays, 106
stripped programs, 17
Sun WorkShop Button Editor, 26
Sun WorkShop Custom Buttons window, 26
Sun WorkShop Debugging Options dialog box, 25
suppress command, 113, 126, 128
suppress_startup_message environment variable, 32
suppression of last error, 127
Surface graph type for array displays, 152
shading choices for, 153
texture choices for, 153
symbol names, qualifying scope, 36
symbol_info_compression environment variable, 32
symbols
choosing among multiple occurrences of, 35
determining which dbx uses, 40
printing a list of occurrences, 39
system event specifications, 75

T

templates
class, 188
stopping in all member functions of, 193
displaying the definitions of, 188, 191
function, 188

instantiations, 188
printing a list of, 188, 190
looking up declarations of, 42
thread command, 171
threads
current, displaying, 171
information displayed for, 170
list, viewing, 172
other, switching viewing context to, 171
printing list of all known, 172
printing list of normally not printed (zombies), 172
switching to by thread id, 172
threads command, 172
trace command, 63
trace_speed environment variable, 32, 63
tracei command, 223
traces
controlling speed of, 63
implementing, 87
listing, 64
setting, 63
tracing at the machine-instruction level, 223
trip counters, 71
troubleshooting tips, runtime checking, 138
turning off
runtime checking, 112
the display of a particular variable or expression, 101
the display of all currently monitored variables, 101
turning on
memory access checking, 111, 112
memory leak checking, 112
memory use checking, 111, 112
types
declarations, looking up, 41
derived, Fortran, 213
looking up declarations of, 41
looking up definitions of, 42
printing the declaration of, 42
viewing, 41

U

uncheck command, 112
undisplay command, 101
unhide command, 95
unintercept command, 185
unsuppress command, 126, 128
up command, 92
updating graphs of data automatically, 156

V

variable type, displaying, 42
variables
and conditional breakpoint, 65
assigning values to, 101, 238
changing after fixing, 165
declarations, looking up, 41
determining which dbx is evaluating, 97
displaying functions and files in which defined, 97
event specific, 85, 86
looking up declarations of, 41
looking up definitions of, 41
monitoring changes, 100
outside of scope, 98
printing the value of, 98
qualifying names, 36
turning off the display of, 101
viewing, 41
verifying which variable dbx is evaluating, 97
viewing
classes, 41
members, 41
the context of another thread, 171
the threads list, 172
types, 41
variables, 41
visiting
a file, 34
functions, 35
by walking the call stack, 36
vitem command, 150

W

walking the call stack, 36, 92
watchpoint event specifications, 73

`whatis` command, 41, 42, 99, 191
`when` breakpoint at a line, setting, 60
`when` command, 60, 70, 240
`when infunction` command, 62
`where` command, 91, 204
`whereis` command, 39, 97, 190
`which` command, 35, 40, 97
`whocatches` command, 185
`.workshoprc` file, 25

X

`x` command, 218